UNIVERSITEIT ANTWERPEN

Combination of Domain-Specific Languages

by

Rafael Ugaz

A thesis submitted in partial fulfillment for the degree of Master of Science: Computer Science

in the
Mathematics and Computer Science
Modelling, Simulation and Design Lab

June 2015

UNIVERSITEIT ANTWERPEN

Abstract

Mathematics and Computer Science
Modelling, Simulation and Design Lab

Master of Science

by Rafael Ugaz

Domain Specific Languages have a central role in the field of model-driven engineering. Combining them provides advantages like re usability and modularity. Language reuse, can bring to the field of model-driven engineering the same benefits it brought to software engineering, such as avoidance of duplication of efforts, which in turn translates to lower production costs. In this thesis, we present an approach for combining DSLs that applies existing model combination techniques. Our approach combines all three of language components of a DSL: abstract and concrete syntax, and semantics. We illustrate our technique by applying it to our running example, a DSL for designing role-playing games as well as two secondary DSLs for shortest path calculation and event listening. A prototype of our technique has been implemented for the AToMPM meta-modelling tool and used for combining the running example of this thesis.

Contents

Abstract **i**

List of Figures **v**

1 Introduction **1**
- 1.1 Context . . . 1
  - 1.1.1 Domain-Specific Modelling . . . 1
  - 1.1.2 Domain-Specific Languages . . . 2
- 1.2 Problem Statement . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Motivation . . . 3
  - 1.4.1 Re-usability . . . 4
  - 1.4.2 Modularity . . . 4
- 1.5 Outline . . . 5

2 Background and Terminology **6**
- 2.1 Directed Graph . . . 6
- 2.2 Model . . . 6
- 2.3 Metametamodel . . . 7
- 2.4 Metamodel . . . 7
- 2.5 Modelling Language . . . 7
- 2.6 Transformation Rule . . . 9
- 2.7 Model Transformation . . . 9
- 2.8 Aspect Oriented Modelling . . . 10

3 Running Example **11**
- 3.1 The Role-Playing Game DSL . . . 11
  - 3.1.1 Abstract Syntax . . . 11
  - 3.1.2 Concrete Syntax . . . 12
  - 3.1.3 Semantics . . . 13
  - 3.1.4 Performance . . . 14
- 3.2 Pathfinding DSL . . . 16
  - 3.2.1 Syntax . . . 16
  - 3.2.2 Semantics . . . 17
- 3.3 Event Listener DSL . . . 19
  - 3.3.1 Syntax . . . 19
  - 3.3.2 Semantics . . . 20

3.4 AToMPM . . . 21
3.5 Code Generation . . . 21

4 Model Combination 22
4.1 Mechanisms . . . 22
4.1.1 Correspondence model . . . 23
4.1.2 Match operation . . . 23
4.1.3 Compose operation . . . 24
4.1.4 Merge operation . . . 24
4.1.5 Extend operation . . . 24
4.1.6 Implement operation . . . 25
4.2 Techniques . . . 25
4.2.1 Merge . . . 25
4.2.1.1 Merging based on correspondences by Pottinger . . . 26
4.2.1.2 Package Merge in UML 2 . . . 27
4.2.1.3 Metamodel merge in GME . . . 28
4.2.1.4 Applied technique . . . 29
4.2.2 Templates . . . 30
4.2.2.1 Template Instantiation by Emerson and Sztipanovits . . . 30
4.2.2.2 Templates in metaDepth . . . 32
4.2.2.3 Applied technique . . . 33
4.2.3 Interfacing . . . 33
4.2.3.1 Applied technique . . . 34

5 DSL Combination in AToMPM 35
5.1 Template DSLs . . . 35
5.1.1 Pathfinding Template . . . 36
5.1.2 EventListener Template . . . 38
5.2 Combination Process . . . 39
5.2.1 Abstract Syntax . . . 42
5.2.1.1 Binding Classes . . . 43
5.2.1.2 Binding Associations . . . 43
5.2.1.3 Binding Fields . . . 44
5.2.2 Concrete Syntax . . . 45
5.2.3 Semantics . . . 47
5.2.3.1 Abstract Rules . . . 48
5.2.3.2 Operations . . . 49
Modeling approach . . . 49
Passing data . . . 50
Nested rules . . . 51
5.2.4 Model Migration . . . 54
5.3 Limitations of the Approach . . . 55

6 Conclusion 57
6.1 Comparison with related work . . . 59
6.2 Future Work . . . 60

A Example Combination in AToMPM **64**

Bibliography **71**

List of Figures

3.1 Abstract syntax of the RPG formalism defined using class diagrams . . . 12
3.2 Partial icon definition of the RPG DSL 13
3.3 Example instance model of the RPG DSL 14
3.4 Operational semantics of the RPG formalism defined using rule-based transformations 15
3.5 Abstract syntax model of the Pathfinding DSL 17
3.6 Schedule model of the Pathfinding DSL defined using the MoTif Transformation formalism 18
3.7 Rule model for the rule block `VisitNeighbor`, shown in the schedule model of Figure 3.6 18
3.8 Abstract Syntax model of EventListener DSL 19
3.9 Concrete Syntax model of EventListener DSL 19
3.10 Operational Semantics model of EventListener DSL 20

4.1 Merge operation of two class diagrams depicting two different metamodels of an RPG 26
4.2 Example of the template instantiation technique in the context of the RPG DSL 31

5.1 The abstract syntax *SimpleClassDiagram* model of the Pathfinding template 36
5.2 The sub-transformation used in the Pathfinding DSL, abstract rules are denoted by a & prefix in their name. 37
5.3 The default implementation of the `&SetSrc` abstract rule of the transformation shown in Figure 5.2. The text in the yellow area is a description of the expected behaviour of the abstract rule (it adds no extra behaviour). 37
5.4 The abstract syntax *SimpleClassDiagram* model of the EventListener Template DSL 38
5.5 The operational semantics of the EventListener Template DSL, the abstract rules are denoted by a & prefix in their name. 39
5.6 FTGPM of the combination process from the point of view of the user . . 40
5.7 Example of a field binding operation 46
5.8 Binding of the `PathfindingTemplate` and the `RPG` languages 47
5.9 The produced abstract syntax model after processing the binding relations of Figure 5.8 47
5.10 Transformation rule of the `getWeight()` operation using session keys. The condition and action code of each pattern block can be seen below it. 51
5.11 Transforming a fragment of a schedule to support an operation rule . . . 53

6.1 First part of the overview of the classification of composition techniques . 62

6.2 Second part of the overview of the classification of composition techniques 63

A.1 combine . . . 64
A.2 Then, the user is asked to select the User DSL metamodel, in this example the *RPG* DSL. . . . 65
A.3 Next, the combination process pre-loads a transformation and asks the user to create the desired binding links (see subsection 5.2.1). When finished, the user can play the already loaded transformation to continue with the combination process. . . . 65
A.4 Here, we see the binding links created, coloured red. We have bound the &*Edge* and &*Node* classes to the Neighbour and Tile classes, respectively. This is done with the intention of supporting finding the shortest path from a given Tile to another. . . . 66
A.5 In this figure we can see the resulting combined metamodel after the *TemplateCombination* transformation has been executed. In this case it replaced the class binding links with inheritance links and it removed the redundant *Src* and *Dst* associations between Tile and Neighbour. The details of this transformation are explained in subsection 5.2.1 . . . 66
A.6 Next, the combined abstract syntax metamodel is compiled into a metamodel file. This file (along with the concrete syntax metamodel file) can be used for loading a metamodel toolbar of the AS which allows the creation of models that conform to the said metamodel. . . . 67
A.7 For combining the concrete syntax, the user is asked to select the desired Template DSL concrete syntax model. . . . 67
A.8 Then, the user is asked to select the desired User DSL concrete syntax model. . . . 68
A.9 With the models selected, they are automatically combined as explained in subsection 5.2.2 and subsequently compiled to a concrete syntax metamodel which, together with the abstract syntax metmaodel, allow the creation of instances of the combined DSL. . . . 68
A.10 The pattern metamodel is also compiled at this point, since it requires both concrete and abstract syntax model in order to be generated. This pattern metamodel is used in the transformation rules used for defining the operational semantics. . . . 69
A.11 Next, the operational semantics models, contained in the *OpSem* folder, are cloned to the folder of the new combined DSL. The models in this folder are also modified, by updating their types, in order to make them compatible with the combined formalism (see subsection 5.2.4). . . . 69
A.12 After these steps, the guided part of the DSL combination is complete. What still has to be done at this point is to implement the operational semantics interface as explained in subsubsection 5.2.3.1. . . . 70

Chapter 1

Introduction

This chapter introduces the context of this thesis, as well as the problems that it addresses and the research questions that it tries to answer.

1.1 Context

The context of this thesis consists in domain-specific modelling and languages.

1.1.1 Domain-Specific Modelling

Domain-Specific Modelling (DSM) [1] is a branch of Model Driven Engineering (MDE) [2] that aims to raise the level of abstraction beyond current programming languages by specifying the solution to a problem in a language that uses concepts specific to the given problem domain. This allows domain-experts (who should not need any programming expertise) to play active roles in development efforts by modelling the solution using only domain constructs that are familiar to them [3].

The final products of a DSM solution are complete artifacts (e.g., executable programs, documentation, test cases or other models) that are (ideally) equivalent to those created using traditional software engineering techniques. These artefacts are automatically generated from their high-level specifications (created by the domain experts) via domain-specific transformations. These transformations are created by more technically experienced programmers and can produce as result either another model (of an intermediate modelling language) or executable code (the final product). This automation minimizes the accidental complexity of the system by avoiding the need to perform the mappings from domain to design and then to code manually. Since this translation is automated, it can also be repeated effortlessly, therefore raising the level of abstraction of the solution to the domain-specific modelling level. All the other abstraction levels of the solution are in this way hidden from the user or domain expert. Thus, DSM effectively shifts the focus of solutions to their design rather than their implementation.

DSM can also provide translation in the other direction, i.e., from executable code to a domain-specific model, via traceability. This is possible because of the creation of *traceability links* between artefacts at different levels of abstraction during the generation stage. Following any given chain of links, a concept can be found at any of the existing levels of abstraction. Traceability has applications for inspecting the system at runtime, debugging, formal analysis results, etc.

The important raise in abstraction of DSM is mainly achieved by the specification of a *Domain-Specific Language*, which is the most external and visible part for the user.

1.1.2 Domain-Specific Languages

Domain-Specific Languages (DSL's) are small and focused modelling languages that allow the specification of systems at a high-level of abstraction.

Modelling languages are essential elements of MDE and are defined by the following three parts:

- The *abstract syntax* defines the internal structure concepts and rules of a language and can also constraint all the possible model instances created with the language to a subset of valid or well-formed ones.
- The *concrete syntax* takes care of the notation used to represent these concepts, be it textually or visually.
- Finally, the *semantics* describes the meaning of the models in terms of a set of known concepts, referred to as the semantic domain.

For a more detailed and formal definition of modelling languages see chapter 2.

1.2 Problem Statement

Because of their focused nature, DSLs are not as effective in representing a whole system, but rather a *view* from it. These views, or *language modules*, need to be combined at some stage in the software development process in order to specify the whole system.

In addition, most of the current methods for their engineering still require them to be built completely from scratch. There is thus a need for mechanisms for the combination of these language modules or fragments.

Additionally, the combination of DSL's would allow the reuse of existing language modules for creating new ones. Language reuse can bring to the MDE realm the same benefits that it brought to the realm of software engineering, mainly the avoidance of duplication of efforts, which in turn can lower the initial cost of building DSLs.

1.3 Research Questions

This work focuses on answering the following main research question:

Is it possible to combine domain-specific languages in a modular way?

We can expand the main research question above and also propose the following sub questions:

1. Can we facilitate the engineering of DSLs via the combination of smaller DSL modules?
2. Can we combine all the language components of DSLs, i.e., syntax (abstract and concrete) and semantics?
3. Can we apply model and metamodel combination techniques, surveyed in previous work [4], for the combination of DSLs?
4. Can DSL combination be generic, i.e., be applicable to models with heterogeneous metamodels?
5. Can we combine DSLs using a model-based approach for the process as well for the parts combined?

1.4 Motivation

In this section we discuss the two main motivations DSL combination more thoroughly.

1.4.1 Re-usability

Most of the time, language engineers have to develop DSLs completely from scratch, using only their own experience and expertise to do so [5]. This is in contrast to general purpose or third-generation programming languages (3GL), where an engineer is able to re-use the existing work of others to speed up the developing process or improve the quality of the product. For example, the Petri nets language can be re-used for creating similar languages that use a variant of the state-transition structure.

For DSLs, composition is one of the missing processes for enabling this feature, along with a more formal specification of them and a library of high-quality and reusable DSLs. In [5], Emerson and Sztipanovits afirm that re-usability can bring to domain-specific modelling the same benefits that software reuse brought to 3GLs:

- Avoidance of duplication of effort (no overlap of concepts)
- Emergence of high-quality reusable DSLs (e.g., Statecharts or Petri nets)
- Recognition of key DSL modelling patterns and best practices
- Significant reduction in the time-to-market of DSLs.

Note that only the first item, avoidance of duplication of effort, is relevant for the scope of this thesis, since it facilitates the development of DSLs.

1.4.2 Modularity

Modularity of modelling languages is a design technique that considers a system as multiple components.

Modularity has benefits similar to those of encapsulation in software engineering, namely a clean separation of, in this case, the models. It also makes the system easier to maintain and extend because of its modular structure as opposed to a tangled one where two tasks become unnecessarily difficult.

But modularity also has the disadvantage of being less efficient performance wise than its counterpart, although this can be remedied by code optimization. The more modular a system is, the more steps are needed to perform a task. A simple example is the access of an object of a module many associations away from another one; in a modular system, a chain of consecutive accessing operations through all modules in between would be needed while in a non modular system this could be done directly with a single operation. An example of a modular language is the Discrete EVent System Specification formalism (DEVS) [6].

When faced with the task of modelling a complex system, breaking it down into a set of modules, where each one deals with a different concern, makes the task much easier. These modules can be defined in different languages, each one better suited for the different view or aspect of the system that it specifies. For instance, the behavioural concern of the system can be designed using State Charts based language while the structural part uses UML Class Diagrams. In the context of our work, we even want to be more specific by, for example, subdividing the behaviour of a system into multiple smaller DSLs. Examples of this approach can be found in the PhoneApps application, a multi-concern DSL, in [7] and in [8], where textual language modules are used to model a traffic light system.

The problem arises when the design phase is finished and it is time to create the final application, which requires the smaller models of the system to be synthesized back together in some way. If the models were created with the same modelling language (i.e., they conform to the same meta-model), a *homogeneous* composition is needed. But if, on the other hand, the models were designed in different modelling languages or paradigms, they require a *heterogeneous* composition. The latter being, as expected, more difficult since it would require a translation mechanism between the languages by, for example, an interface.

1.5 Outline

This thesis structure is divided in the following way. In chapter 2 we provide a set of basic background concepts and terminology which are used throughout the thesis. In chapter 3 the running example is introduced: a DSL for creating role-playing games. Two secondary DSLs are also presented in this chapter. These DSLs are used to illustrate the combination technique presented in the thesis. Next, chapter 4 presents the model combination mechanisms and techniques that have been adapted to fit our solution. In chapter 5, the implemented solution for combining of DSLs is displayed. A walk through the process is described as well as the limitations of the approach. Finally, chapter 6 recaps the work done in the thesis with conclusions about the research questions presented in this chapter and a discussion about related and future work.

Chapter 2

Background and Terminology

In this chapter we define several concepts and present terminology that is used throughout the rest of the paper. This chapter is based on the set of definitions for model composition frameworks presented by Bézivin et al. in [9], [10] and [11].

2.1 Directed Graph

A formal way to define models, which is also useful for their visualization, is to represent them as *directed graphs*. Let $G = (N_G, E_G, \Gamma_G)$ be a directed multi-graph that consists of three elements:

- N_G, a finite set of nodes;
- E_G, a finite set of edges;
- $\Gamma_G : E_G \to N_G \times N_G$, a function that maps edges to their source and destination nodes.

2.2 Model

A *model* $M = (G, \omega, \mu)$ is a triple where:

- $G = (N_G, E_G, \Gamma_G)$, is a directed multi-graph;
- ω is a model (called the reference model) with an associated graph $G = (N_G, E_G, \Gamma_G)$;

- $\mu : N_G \cup E_G \to N_\omega$, is a function that maps all the nodes and edges of G to nodes of G_ω. This means both nodes and edges of G are constrained by nodes from G_ω (which is the graph of its reference model).

The relation between a model and its reference model is called *conformance*. Thus we say that a model *conforms* to its reference model.

This definition allows an indefinite chain of models that conform to each other. However, three levels are usually sufficient. We call these three levels metametamodel (M3), metamodel (M2) and terminal model (M1).

2.3 Metametamodel

A *metametamodel* is a model that is its own reference model, in other words it conforms to itself. This concept is called *metacircularity* and it avoids endless "meta" prefixes.

2.4 Metamodel

A *metamodel* is a model such that its reference model is a metametamodel.

2.5 Modelling Language

A modelling language is a language that defines a set of consistent rules for expressing information or knowledge in the form of models. In [12], Chen et. al give a formal definition of a Domain-Specific Modelling Languages. This definition can also be applied to the broader concept of modelling languages. They formally define a modelling language as a five-tuple of the form:

$$L = \{C, A, S, M_S, M_C\}$$

Where,

1. C is the concrete syntax, that defines the notation used to express the models which may be graphical, textual or mixed;
2. A is the abstract syntax, which defines the concepts, relationships and constraints or well-formedness rules of the language;

3. S is the semantic domain, a set of concepts (usually defined in some formal framework) in terms of which the meaning of the models of the language are explained;
4. $M_C : C \to A$ is the syntactic mapping, it assigns a syntactic construct of the concrete syntax C to the elements of the abstract syntax A;
5. $M_S : A \to S$ is the semantic mapping, which relates the syntactic elements of the abstract syntax A to those of the semantic domain S.

A definition of a modelling language can be therefore divided into the following three parts, where each one can be specified using models:

Abstract Syntax

The abstract syntax describes the concepts of a language and their properties, the legal connections between them, the model hierarchy structures, and also grammatical rules that enforce model correctness. These rules can significantly reduce the possible design space which helps in designing correct applications.

The abstract syntax of a modelling language is normally specified with a meta-model [13]. The prefix "meta" is used to denote that an operation, in this case modelling, is applied twice. In other words, a meta-model is a conceptual model of a modelling language. A meta-model provides a formal specification of the language which, supported by tools (e.g., AToMPM, MetaEdit+), is used to create and modify models as well as generating code from them.

Concrete Syntax

The concrete syntax provides a representation of the abstract syntax of a meta-model as a mapping (syntactic mapping) between the meta-model concepts and their textual or graphical representation. A language can have several concrete syntaxes.

Semantics

The semantics of a modelling language define the meaning of the syntactically correct concepts of a modelling language. The semantics of a language comprise the already defined concepts of semantic domain and semantic mapping.

There are many approaches for defining the semantics of a language. For this work, however, the following two are more relevant:

- *Operational* semantics often encode behaviour and give meaning to a language by describing the transformation of the system from one state to the next (possibly along some time model).

  They are usually better suited for giving meaning to behavioural languages (e.g., finite state automata (FSA), Petri nets or activity diagrams). Since structural languages (e.g., class diagrams or entity relationship diagrams) have a more static and non-behavioural nature, they cannot be very well represented by this type of semantics;

- *Denotational* semantics define the meaning of a language in terms of another language or formalism for which well defined semantics (operational or denotational) exist, for example code, mathematics or Petri nets. With this kind of semantics, any type of language (behavioural or structural) can be specified, as long as the language it is being defined in can properly represent its concepts.

2.6 Transformation Rule

A transformation or *graph* rule $R : M_{IN} \rightarrow M_{OUT}$ is a function that takes a model M_{IN} as input, performs a matching of the LHS subgraph over it, replaces the match with another subgraph RHS and outputs the model M_{OUT}. A graph rule is parametrized by:

- LHS, a left-hand side pattern that should be present in the input model;
- RHS, a right-hand side pattern that describes how the LHS should be transformed after applying the rule;
- NAC, a negative application condition pattern that should not be present in the input model for the rule to be applicable.

2.7 Model Transformation

A model transformation T, is an operation with signature $S_{OUT} = T(S_{IN})$ that takes a set of input models S_{IN}, executes a set of transformation rules S_R over the model elements and produces a set of models S_{OUT} as output.

A transformation is itself also a model and therefore all the general operations applicable to models may be applied to transformations (including transformations that are called higher-order transformations).

2.8 Aspect Oriented Modelling

Aspect Oriented Modelling (AOM) translates the concepts and ideas applied at the code level by aspect oriented programming (AOP) [14] to the model level. Usually, a system contains one or more so called cross-cutting concerns spread along the system (e.g., Logging or Debugging). These cross-cutting concerns are difficult to maintain when tangled all over the system. AOM provides a way to isolate these concerns into centralized and easy to maintain models which are called *aspect* models. The way to keep these aspects where they should be located in the system is by defining two core concepts inside them: a *pointcut* and an *advice*. The pointcut defines all the joinpoints where the cross-cutting concern is located in the system. And the advice defines what functionality, either a code segment in AOP or model instances in AOM, to insert at the join-points. The act of combining aspect models into a base model is called *weaving*. This terminology can be used in different contexts or implementations, for example when using rules for weaving, a rule can be considered as the aspect, where the LHS is the pointcut, the RHS is the advice, and the match is the joinpoint.

Chapter 3

Running Example

In this chapter, the RPG domain-specific language is introduced. In previous work [15], a complete DSM solution has been implemented for this language, which includes a code generator for Android applications as well as a domain framework. This language will be used as the running example to illustrate the combination techniques that are applied in the following chapters. In this chapter we also present two other DSLs, *Pathfinding* and *EventListener* which will be later used to extend the RPG DSL with new features.

3.1 The Role-Playing Game DSL

The Role-Playing Game language, as its name indicates, allows the modelling of Role-Playing Games (RPGs). An RPG is a game where players assume the roles of characters in a fictional setting. The games created with this DSL fall more into the sub-category of RPGs called *dungeon crawlers*. As the name indicates, the player has to explore a dungeon taking the role of a character. In the dungeon the player encounters enemies that it has to fight, items to collect and quests to complete.

3.1.1 Abstract Syntax

The abstract syntax of the RPG consists of all the structural elements that take part in the game. It has been modelled using the `SimpleClassDiagrams` language as it can be seen in AToMPM. To verify that a model is well formed, the abstract syntax is also enriched with constraints such as cardinalities.

The abstract syntax model can be seen in Figure 3.1. At the top most level, the game contains one or more *scenes* or “levels” where all the other structural elements are

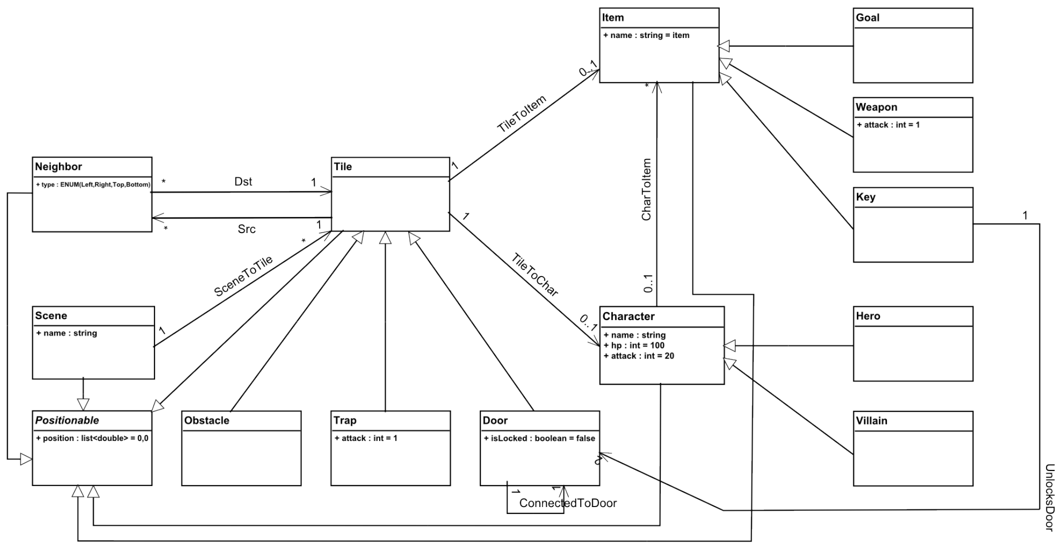


Figure 3.1: Abstract syntax of the RPG formalism defined using class diagrams

contained. In each scene, there are a number of *tiles* that can be connected to each other from the left, right, top or bottom. This way, a two-dimensional map is created for each scene. The game also contains characters, a hero and villains. There can be exactly one hero but multiple villains. All characters can stand on one tile at the same time and a tile can hold at most one character. There are four different types of tiles: an *obstacle* tile on which no character can stand, a *door* tile, which is connected to another door tile and can be used by a character to move form one scene to the other, a *trap* tile, where a character loses life points whenever it stands on it and a regular tile, with no added effects. On a regular tile, there can be an *item*. There are three types of items: *goals*, can be picked up by the hero (its purpose is explained in the semantics), *weapons*, that when picked up give a bonus in attack to the character and *keys*, which are used to unlock doors (and enable a door for use).

3.1.2 Concrete Syntax

In AToMPM, concrete syntax is specified using a the Icon Definition language. This is a simple language that for defining one-to-one mappings between each abstract syntax entity and its visual representation. A model conforming to this language and used to define the concrete syntax of the RPG DSL is illustrated in Figure 3.2. An instantiated RPG model, using the visual concrete syntax in conjunction with its abstract syntax is shown in Figure 3.3.

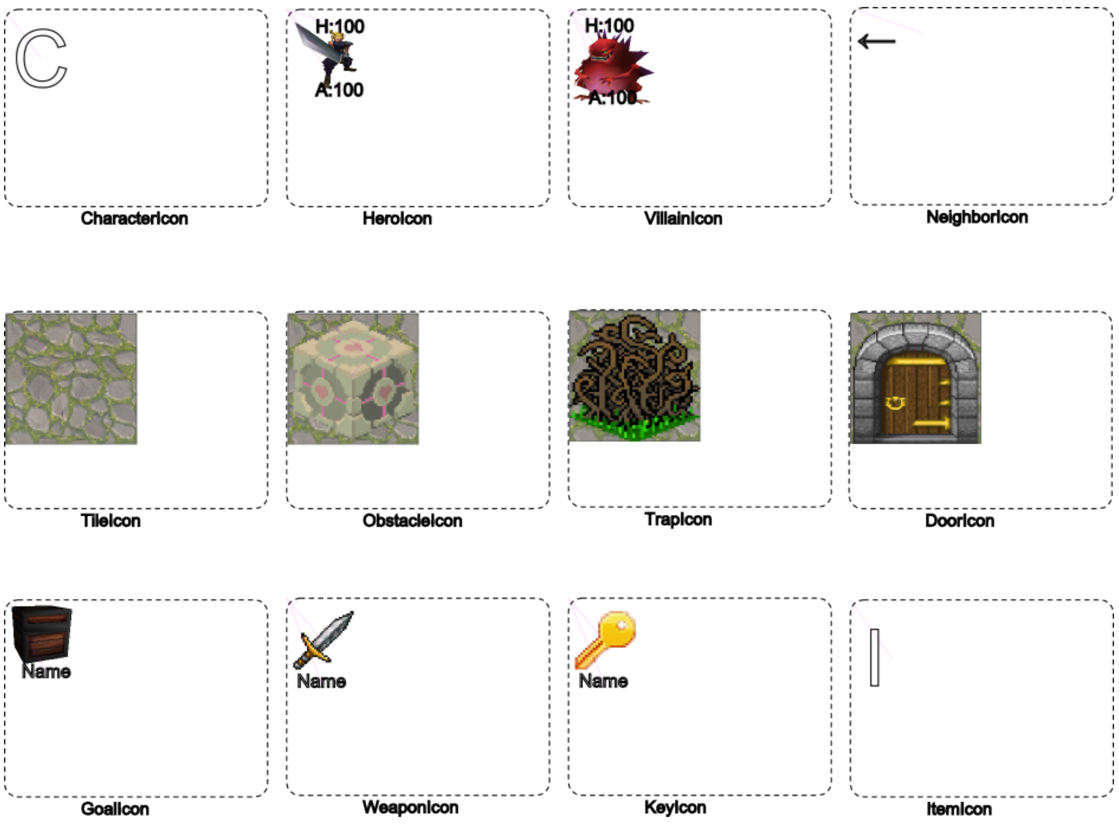


FIGURE 3.2: Partial icon definition of the RPG DSL

3.1.3 Semantics

Since role-playing games have behaviour, their semantics are of the type *operational*. They describe explicitly how the game can be executed or *simulated* and have been defined using the `TransformationRule` and `MoTif` [16] transformation languages.

The operational semantics can be described with the following rules:

- The game is turn-based, first the hero gets a turn to either move or attack, then all the villains get a turn to do the same, then it goes back to the hero and the cycle repeats itself until the end game conditions are met.
- A character can move from one adjacent tile to another, as long as the tile is not an obstacle tile and is not occupied by another character.
- A character can attack an enemy that stands in any adjacent tile.
- An item can be picked up by the hero by walking on its tile. Every item can only be picked up once.
- If the hero picks up a weapon, the damage of the weapon is added to the hero's attack.

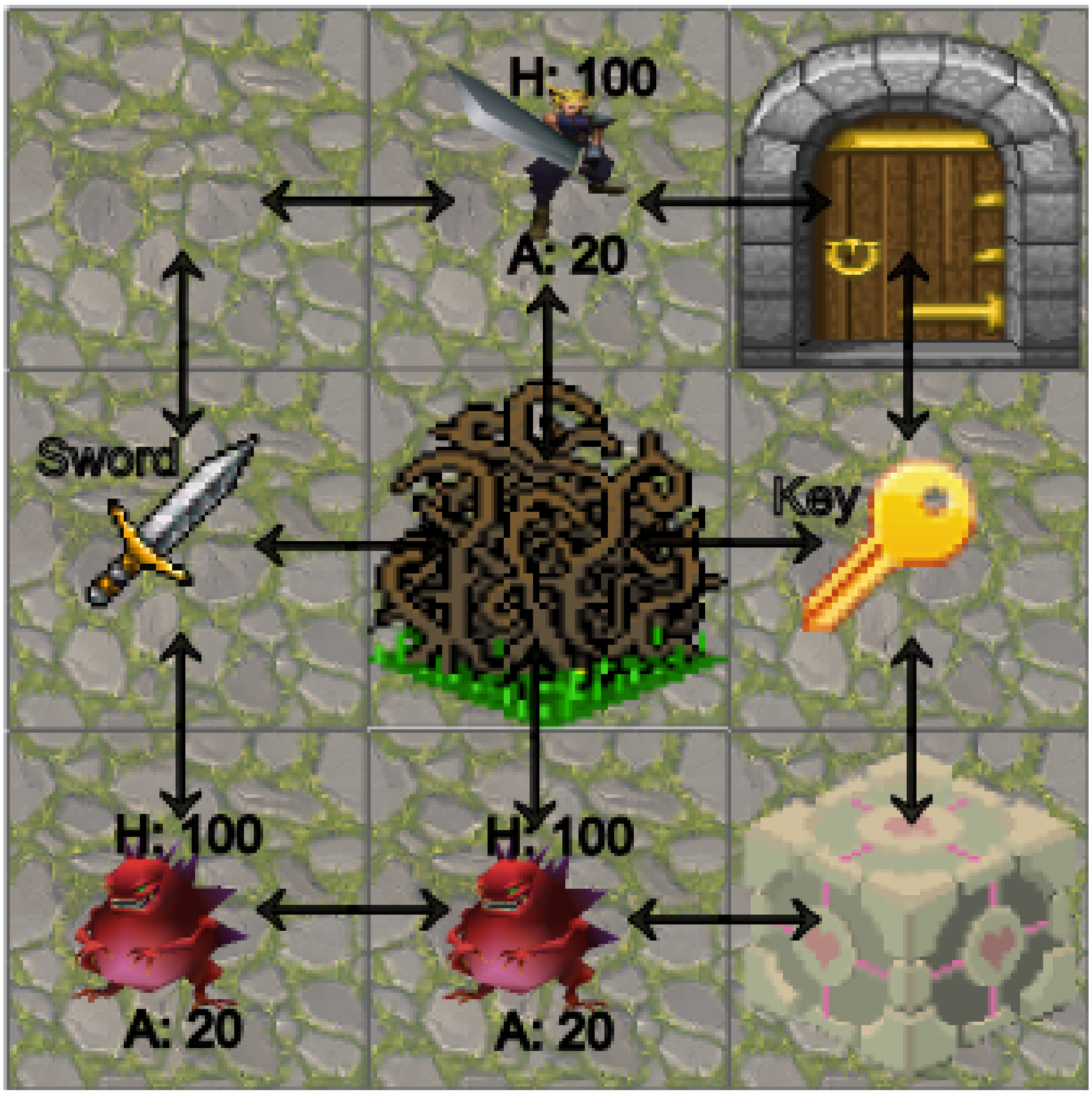


FIGURE 3.3: Example instance model of the RPG DSL

- If the hero picks up a key, he can use a door that is unlocked by that key.
- The player wins the game if he picks up all the goals. There must be at least one goal at the beginning of the game.
- The player loses the game if the hero is killed.

The semantics model designed for this DSL can be seen in Figure 3.4.

3.1.4 Performance

When executing the operational semantics in the AToMPM client, the game runs very slowly, however when running it in an Android device using the Java generated code and framework rule matcher, it performs much better. This is related to the architecture of

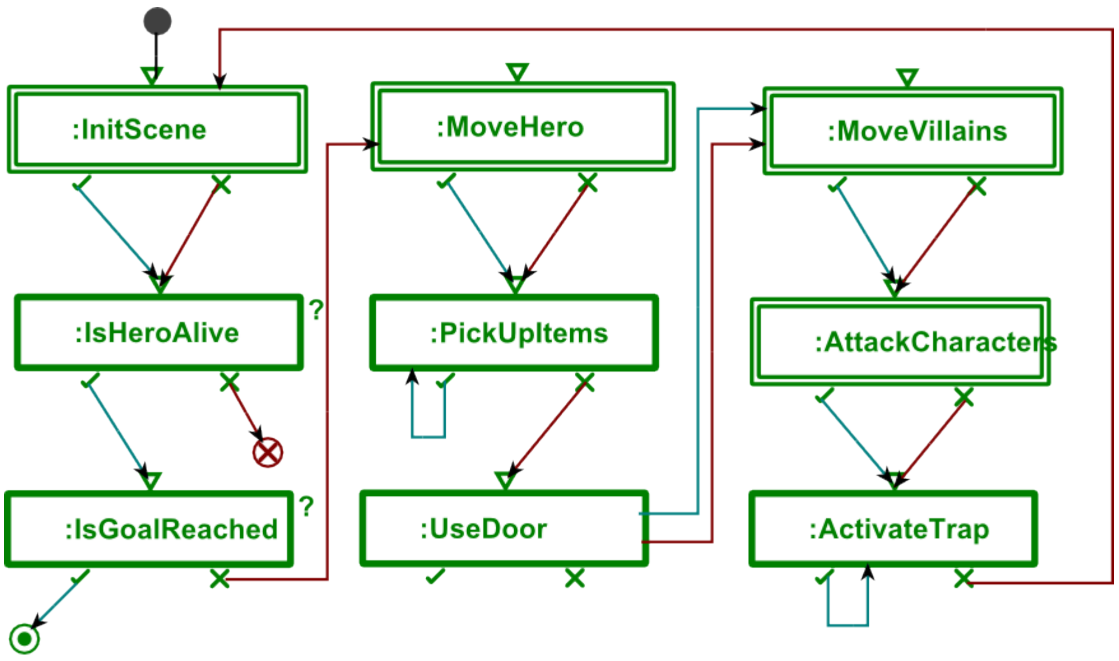


FIGURE 3.4: Operational semantics of the RPG formalism defined using rule-based transformations

AToMPM which depends on the network communication between the client application (with whom the user interacts) and the server, where all the operations (including rule transformations) are executed. Although modifying the architecture of AToMPM is outside of the scope of this thesis, the performance can still be improved with the following optimizations to the transformation rules:

- Having multiple **attack rules** for the attack action, e.g., AttackLeft, AttackRight, etc, is more efficient than a single Attack Rule. The current single attack rule used in the transformation has a search space that includes all the tiles in the map while this search space could be greatly reduced by having individual rules for each direction, that way, the rule matching algorithm would only consider as candidates the tiles situated at the appropriate direction of the character.

- The AToMPM rule matcher allows the use of **pivots** in the transformation rules, although this has not been implemented yet in our Java rule matcher. Pivots could speed up the game by reducing the search space that some rules have to go through. For example, all rules that use the node of the Hero in their application (e.g., MoveHero, PickUpItems, IsHeroAlive) have to search for the Tile node where the Hero stands from all the possible Tile nodes. Using pivots, after one of these rules have found this Tile, it can mark it as a pivot and send it to the other rules that use it, avoiding the duplicate searching of this Tile by these other rules.

3.2 Pathfinding DSL

The Pathfinding DSL enables designers to create directed graphs and calculate the shortest path between any two nodes of the graph.

The pathfinding DSL allows the creation of a weighted graph with a single source and destination nodes. After creating such a graph, the operational semantics of the DSL can be executed in order to find the *shortest route* between two nodes. The calculation of the shortest path is based on Dijkstra's algorithm, which has been completely modeled using rule-based model transformations.

The purpose of this DSL is to extend the RPG DSL with pathfinding capabilities. Pathfinding is at the core of any kind of artificial intelligence (AI) in games. It allows to find paths between any two coordinates of the game world. An example of a feature possible because of pathfinding is to have a villain move towards a Hero to attack it and *chase* it. This is a big improvement compared to the default random movement behaviour of the RPG DSL.

3.2.1 Syntax

The abstract syntax of the pathfinding formalism consists of the following elements:

- A *Node* that represents the entity of the same name in the search graph. For our concrete case this is a location in the game map, e.g., a Tile. A node can be a regular node or two special types: source and destination. There must be exactly one source and one destination in the graph for the algorithm to work correctly. The Node also contains attributes required for the Dijkstra's algorithm like the *distance* to the destination node and a *isVisited* flag.
- An *Adjacent* edge denotes that two nodes are connected in the search graph.
- A *Previous* edge which is used after the application of the algorithm to point to the shortest path found. The path can be obtained by navigating from the destination towards the source following the Previous edges.

The abstract syntax of the Pathfinding formalism, like the RPG formalism, has been defined using the `SimpleClassDiagram` language in AToMPM.

The concrete syntax of the Pathfinding formalism is similar to the concrete syntax of an undirected graph with the addition of annotations for specific attributes of Dijkstra's algorithm, e.g., distance, edge weights, and whether a node is visited.

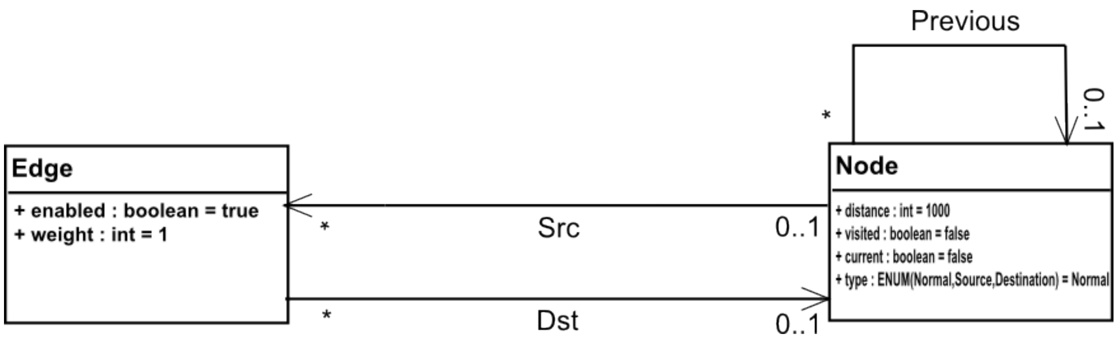

Previous
*
0..1
Edge
+ enabled : boolean = true
+ weight : int = 1
Node
+ distance : int = 1000
+ visited : boolean = false
+ current : boolean = false
+ type : ENUM(Normal,Source,Destination) = Normal
*
Src
0..1
*
Dst
0..1

FIGURE 3.5: Abstract syntax model of the Pathfinding DSL

3.2.2 Semantics

The operational semantics of the Pathfinding formalism model Dijkstra's algorithm completely, including its calculation of the node with the minimum distance.

The semantics of Dijkstra's algorithm can be divided in the following steps:

1. Assign to every node an initial distance value: *zero* for the source node and *infinity* for all other nodes (**:InitNodes**);
2. Mark all nodes as *unvisited.* Set the initial node as *current.* Create a set of the unvisited nodes called the unvisited set consisting of all the nodes (**:InitNodes**);
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has weight 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value(**:VisitNeighbor** and **:VisitNeighborVar**);
4. When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the unvisited set (**:MarkAsVisited**). A visited node will never be checked again;
5. If the destination node has been marked *visited* (**:isCurrentDest**)—when planning a route between two specific nodes— or if the smallest tentative distance among the nodes in the unvisited set is infinity (**:isCurrentInfinity**)—when planning a complete traversal this occurs when there is no connection between the initial node and remaining unvisited nodes— then stop. The algorithm has finished;
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3 (**:FindMinNode**).

These steps have been modeled using the MoTif transformation formalism in conjunction with the rule-based transformations formalism to produce a main schedule containing two sub-schedules (denoted by double edged rectangles) and around 15 rule models. The main schedule model can be seen in Figure 3.6.

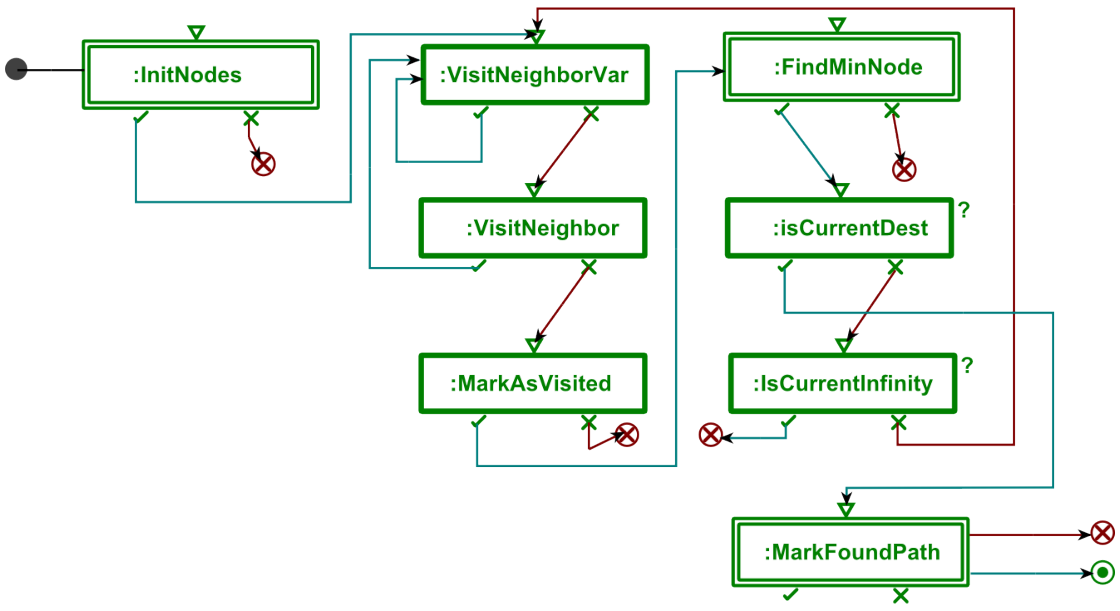


Figure 3.6: Schedule model of the Pathfinding DSL defined using the MoTif Transformation formalism

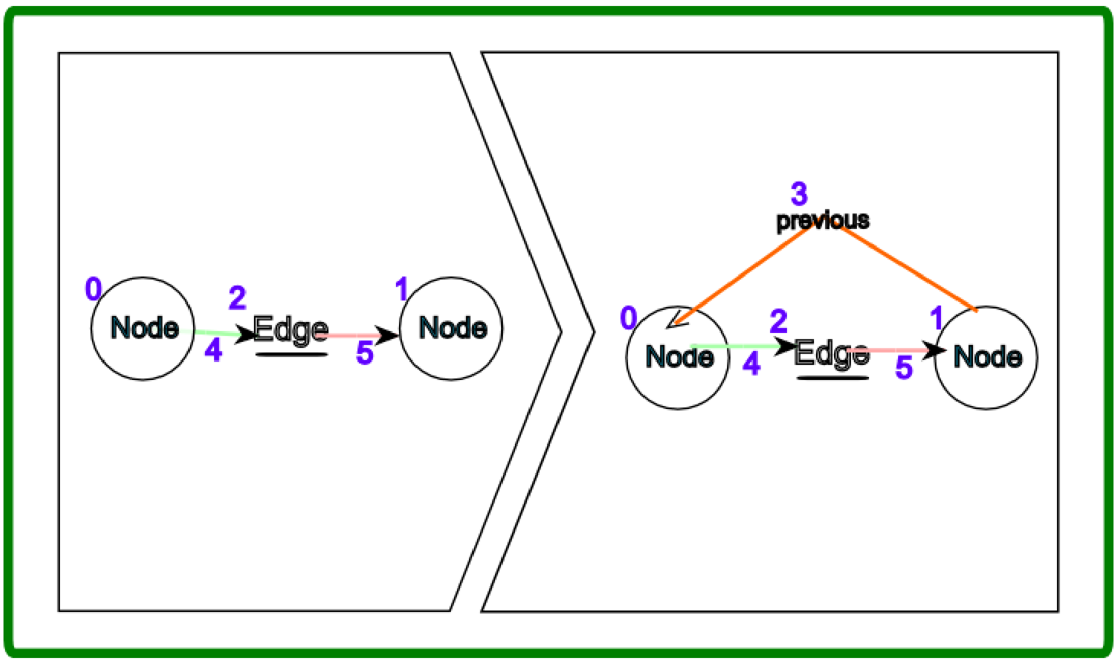


Figure 3.7: Rule model for the rule block `VisitNeighbor`, shown in the schedule model of Figure 3.6

3.3 Event Listener DSL

The *EventListener* Template DSL is a simple language that follows the Observer pattern and is used for handling the key presses sent to the subject (e.g., the tool AToMPM) by the environment.

3.3.1 Syntax

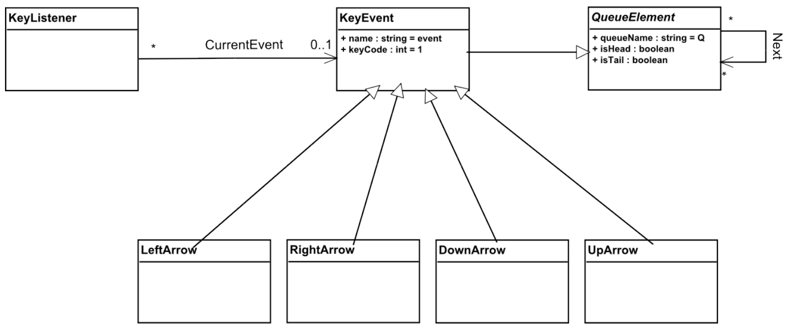


Figure 3.8: Abstract Syntax model of EventListener DSL

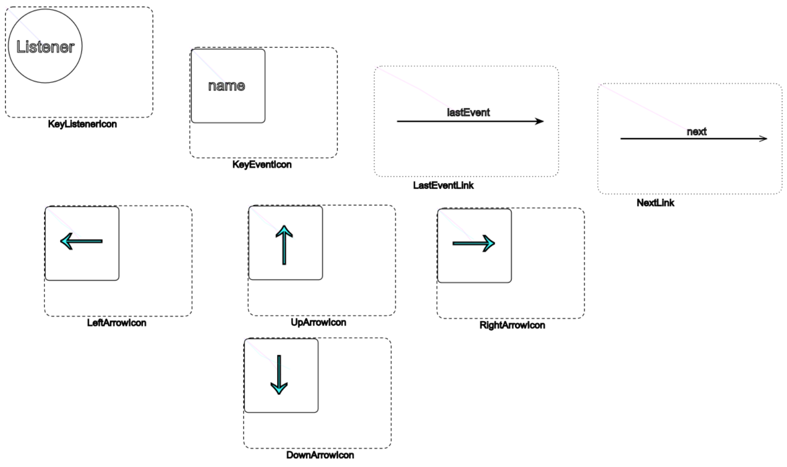


Figure 3.9: Concrete Syntax model of EventListener DSL

The abstract syntax of the DSL primarily consists of the two types *KeyListener* and *KeyEvent*. A *KeyListener* object can be related to at most one *KeyEvent* object, via a *CurrentEvent* link. The *KeyEvent* class can be of several subtypes depending on the key press that it represents. This can be seen as redundant since the type of the key pressed is already stored in the *keyCode* attribute but modelling this characteristic explicitly is useful for making rule transformations more clear as well as interactions with the AToMPM environment. The abstract class *QueueElement* is a helper class that provides the basic functionalities of a queue data structure. We let *KeyEvent* inherit from *QueueElement* to allow the ordering of event objects in a First-In-First-Out (FIFO) manner. The abstract syntax of the DSL, containing all the mentioned elements, can be seen in Figure 3.8. The concrete syntax is shown in Figure 3.9.

3.3.2 Semantics

The semantics of the DSL perform the following tasks in one iteration:

- First, the event at the head of the event queue (the oldest) is connected with the *KeyListener* via a *CurrentEvent* link. This step is performed in the *ProcessEvent* rule;
- Next, the type of the event is checked. Currently we support four types of events, one for each arrow key in the keyboard. This check is done via the *IsXXXArrow* query rules. If the type is not recognized, the transformation fails;
- According to the recognized type of the event, its corresponding action is invoked via the *XXXAction* rules. If any of the action rules fail to apply, the whole transformation fails as well;
- Finally, the current event is deleted, and the event queue is updated appropriately, by making the following event the head of the queue.

The operational semantics, modelled using the MoTif scheduling language are shown in Figure 3.10.

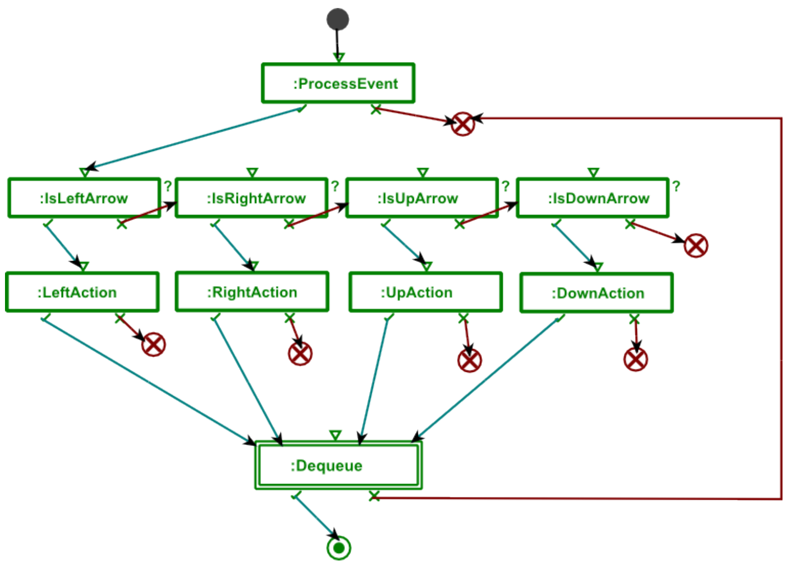


FIGURE 3.10: Operational Semantics model of EventListener DSL

3.4 AToMPM

All the DSLs used in this work have been created using AToMPM (A Tool for Multi-Paradigm Modelling), [17] an open source framework for designing DSL environments, performing model transformations, manipulating and managing models. It is primarily a graphical modelling environment but it is also possible to write textual commands for more advanced users. Model transformations can be explicitly modelled in AToMPM, this is done by defining a left-hand side (LHS), right-hand side (RHS) and negative application conditions (NACs), similarly to graph transformations. The patterns inside the rules use a similar concrete syntax as that of the input and output languages, but adapted to rule patterns.

3.5 Code Generation

A complete DSM solution has been implemented (see [15] for details) along this DSL which allows the automatic generation of instances of RPG that can be simulated in an Android device.

As mentioned before, in a previous work [15], a complete DSM solution was developed for supporting the automatic creation of Android applications (implemented in Java) from models created with AToMPM. This framework supports languages that have been defined using Class Diagrams for their abstract syntax and Rule-Based Transformations for their semantics. A transformation engine was implemented in Java in order to execute the transformations in the Android platform.

Chapter 4

Model Combination

In this chapter we explain model combination in detail. First, we define the fundamental mechanisms present in model combination in a formal way. Next, we describe the main techniques that apply these mechanisms in works created by different authors.

Although the focus of this thesis is how to combine DSL's, this task can also be tackled by using model combination techniques. In [10], Kurtev et al discuss the use of model-based solutions for defining DSL's. They define a DSL's as a set of coordinated models, where each one defines a language components: the abstract syntax, concrete syntax and semantics. This means that model combination solutions can be applied to DSL combination as well.

This relation between models and DSLs also occurs in the tool that will be used throughout this work, AToMPM [17], where DSL's are created by modelling these three components separately. The action of combining two DSLs can therefore be translated to the combination of the three models that define their components, using the model combination techniques explained in this chapter.

Some techniques described in this chapter refer to metamodel combination, this is how the authors of these techniques have labelled them. We consider them in a broader sense *model* combination techniques, since a metamodel has a metametamodel and in the perspective of this metametamodel, the metamodel is a model.

4.1 Mechanisms

In this section we provide a formal definition for the mechanisms that are employed in model combination. It builds upon the definitions provided in chapter 2.

4.1.1 Correspondence model

A correspondence model $C = (G_C, \omega, \mu)$ is a model (section 2.2) that consists of links between elements of different models, such that:

- Let $S = \{M_i = (G_i, \omega_i, \mu_i);\ i = [1..n]\}$ be a set of different models;
- G_C has only two types of nodes: *links* and *link endpoints*;
- For each link endpoint in G_C, there is a link connected to it through an edge;
- Each link endpoint in G_C refers to an element e (node or edge) of a model M_i of the set S;
- Links can have a many-to-many multiplicity as long as they have at least one link point referring to each model.

Correspondence models can be used in model combination for indicating the elements of two different models that have to be combined in some way, e.g., by a merge or extension operation, see subsection 4.1.4 and subsection 4.1.5. They can also be used for traceability between equivalent models at different levels of abstraction, see subsection 1.1.1.

The process of creating a correspondence model is encapsulated in a match operation.

4.1.2 Match operation

The match operation $C = Match(S)$ takes as input a set of models $S = \{M_i = (G_i, \omega_i, \mu_i);\ i = [1..n]\}$, searches for equivalences between their elements and produces a correspondence model C as output. This step defines which elements of one model are to be combined with those of another model. It is possible to automate this operation although it is usually more effective if it is performed manually by a user with knowledge of the domain.

The semantics of the match operation consist of a mix of comparison and conformance rules.

- Comparison rules determine *syntactic* similarities between model elements.
- Conformance rules determines if syntactically similar elements are also *semantically* compatible.

The matching operation is also subject to some basic constraints such as typing and multiplicity. The former requires the matched elements to be of compatible types, i.e., either the same type or one a subtype of the other. The latter applies for the matching of links, in this case the multiplicities on both sides of one of the links should be entirely contained in those of the other link, or vice versa. An example of a violation of the multiplicity constraint is a link with 0..1 and another with 1..$*$ (on the same side) being matched together.

4.1.3 Compose operation

The compose operation $M_{AB} = Compose(M_A, M_B, C_{AB})$ takes two models M_A and M_B and a correspondence model C_AB between them and combines their elements into a new output model M_{AB}.

This operation is considered in this paper as a general and even abstract operation that all the other composition operations must specialize or instantiate. For this operation and its specializations, it is common that the models involved in the process conform to the same metamodel. If this is not the case, an interface between the (meta)models would be required.

4.1.4 Merge operation

The merge operation $M_{AB} = Merge(M_A, M_B, C_{AB})$ takes the same parameters as the more general compose operation. It produces a different output M_{AB} which includes all the elements from M_A and M_B. Merge is a special case of model composition, since it imposes the extra constraints of information preservation, which requires that all the information from the input models should be present in the output models, without duplicate information. The correspondence model is created by the match operation and it specifies which elements are to be merged.

4.1.5 Extend operation

The extension operation $E_{AB} = Extend(M_A, M_B, C_{AB})$ is a special case of the compose operation where:

- The models have the same metamodel
- Its main requirement is to create at least one new **edge** in the resulting model from an element $m_A \in N_{G_A} \cup E_{G_A}$ to an element $m_B \in N_{G_B} \cup E_{G_B}$.

- The correspondence model defines between which elements these new edges are created. Note that in this case the correspondence model does not represent equivalency between elements (as it was the case for the merge operation).

4.1.6 Implement operation

The implement operation $I_{AB} = Implement(M_A, M_B, C_{AB})$ is a special case of the compose operation where:

- It creates at least one **node** in the resulting model that did not previously exist and does not strictly belong to either of the input models M_A and M_B.
- The correspondence model defines links between the *place holder* and the *concrete* elements that implement them. In this case, as in subsection 4.1.5, the correspondence model does not represent equivalence.

Place holder elements in a model represent concepts with missing information. This information has to be provided by a *concrete* element (by implementing it) that satisfies the place holder constraints, e.g., matching its type or signature. This type of elements are generally used for behavioural models, where the missing information is a specific behaviour of the operational semantics.

4.2 Techniques

In this section we go over several approaches by different authors where they apply the mechanisms explained in the last section for the actual combination of (meta)models.

4.2.1 Merge

Model merging is a combination technique that does not assume an unbalanced relationship between its participants, i.e., it is performed on two *peer* models. In terms of set theory it can be expressed as the duplicate free union of two sets, where a set represents the elements of a model.

In Figure 4.1, an example can be seen of a general merge operation on two metamodels depicting different facets of a Role Playing Game (RPG), both defined in the UML class diagrams metametamodel. This simple example contains two different join points or correspondences which are located between the classes Tile and Hero of the same name

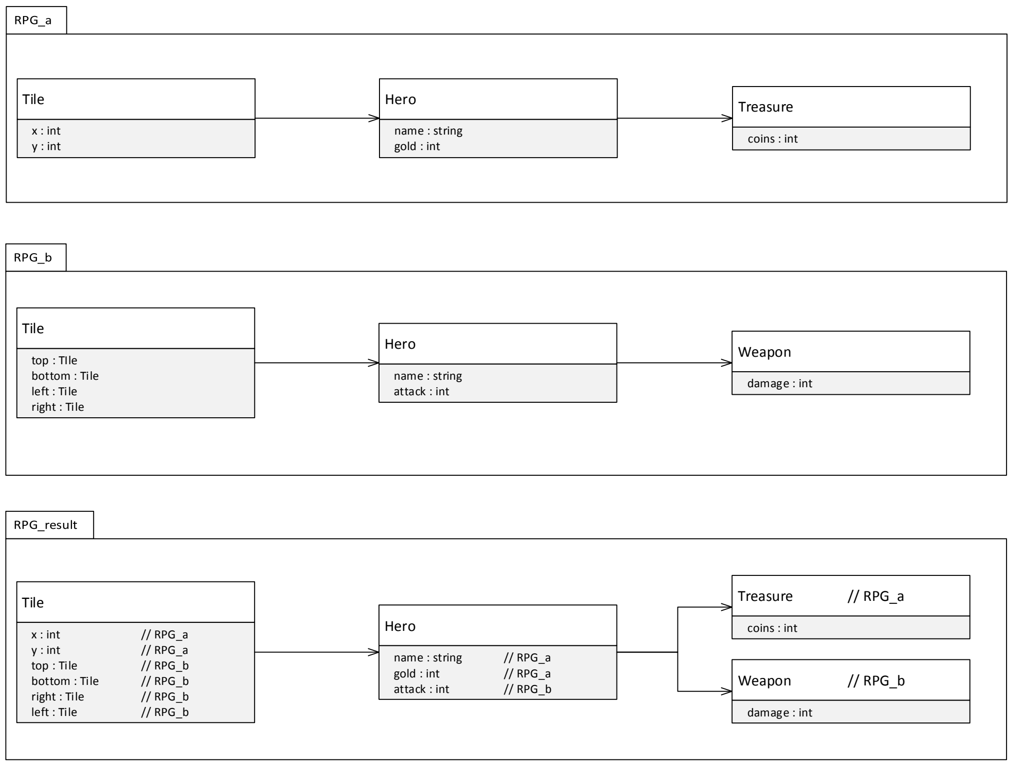


FIGURE 4.1: Merge operation of two class diagrams depicting two different metamodels of an RPG

in both metamodels. The remaining classes Treasure and Weapon are not included in the correspondences and thus they are both included in the final metamodel AB (along with their corresponding associations). Finally, the Tile class of the resultant model AB, an unresolved conflict can be observed. This conflict is caused by the fact that both input metamodels have two different ways of representing the concept of location of a Tile. Model A does this with the x and y coordinates while model B with the neighbouring tiles of the four cardinal directions. This conflict is not properly solved in this example since both concepts have been included in the final model and its solution would require a decision by a user.

In this section, we review the most relevant techniques that apply model merging in practice.

4.2.1.1 Merging based on correspondences by Pottinger

In [18], Pottinger and Bernstein propose an approach to model merging that depends on a set of user-defined correspondences between the models, which compose the *correspondence model*. The authors provide a *generic* framework that can be used to merge all types of models, regardless of their metamodels.

For the *matching operation*, this technique first identifies which elements of each metamodel will be merged together. This information is then stored in the correspondence model composed of a set of correspondences between them. The elements which are not included in the set of correspondences are simply added to the result model without modifying them, similarly to a union operation.

With the *correspondence model* created during the *match operation*, the *merge operation* can begin. This composition is not as simple as the set union since the detection of duplicates (which depends on what is defined as a duplicate, i.e., its semantics) as well as their removal can be complex. In addition, the resulting merged model can also present constraint violations, or *conflicts*, that the merge operation must resolve.

Regarding conflicts, the authors provide the following categorization for those that can arise, based on the meta-level where they occur:

- Representation conflicts occur at the model level and are caused by conflicting representations of the same modelled (real world) concept. An example is having a model A which represents the concept of name by one element *Name* while model B represents it by two elements: *FirstName* and *LastName*. How the concept should be represented in the final model is a decision that is application dependent and is performed before the merging algorithm.
- Meta-model conflicts, are caused by the violation of the constraints of one of the two metamodels involved in the merging.
- Fundamental conflicts are caused by violations of constraints at the metametamodel level, the representation to which all models must conform to. In other words when the result of a Merge would not be a model due to violations of the metametamodel.

4.2.1.2 Package Merge in UML 2

UML 2 [19] defines Package Merge as an operation for merging the contents of two packages together. The elements of the models or metamodels contained in the packages are merged if they share the same name and signature. Package merge aims at allowing the definition of metamodels in UML more modular. It is a directed relationship between two packages which indicates that the contents of the target package are merged into the contents of the source package. The composition occurs in two phases which apply a set of constraints and transformations [20]. First, during the *match operation*, the constraints are used to check if two elements are equivalent. When this is the case, the transformations take care of the actual merging. Elements that do not have a matching counterpart are simply carried over to the result model. Constraints and transformations are expressed declaratively through match rules and transformation rules. These rules are pre-defined for each metamodel type (class diagrams, sequence diagrams, etc) of the UML family of metamodels and are thus not generic.

An important feature that is mentioned in the UML 2 specification and that is very important for all composition approaches is that a resulting element will not be any less capable than it was prior to the merge, in other words that the merge operation does not violate Liskov's substitutability principle. In this case it means that the resulting navigability, multiplicity, visibility, etc will not be reduced as a result of a package merge.

According to Vallecillo [21], the problem with this approach is that its current definition is neither precise nor sound and it does not consider possible conflicts between the structural constraints of the metamodels merged . As a result, it may break the well-formed rules (constraints) of the models that it combines. Furthermore, no traceability links are created between the resultant metamodel and the input models. Finally, he points out the fact that this technique works exclusively for models defined with the UML metamodel.

4.2.1.3 Metamodel merge in GME

In [22], Lédeczi et al. propose an extension to UML that allows metamodel composition from existing metamodels. This extension consists in the addition of three new UML operators for using in metamodel combination:

- The *equivalence* operator is used to denote a full union between two UML class objects. The union includes all attributes and associations, including generalization, specialization, and containment, of each individual class. It can be thought of defining the join points for the source metamodels, similarly to the correspondences definition combined with the merging steps of Pottinger's approach (see subsubsection 4.2.1.1).
- With the *implementation inheritance* operator the child class inherits all of the parent class's attributes, but no associations except the containment ones where the parent functions as the container.
- While the *interface inheritance* operator allows no attribute but does allow full association inheritance, with the exception of containment associations where the parent functions as the container.

The union of the implementation and interface inheritance is the normal UML inheritance and their intersection is null. The three operators mentioned are simply a notational convenience or syntactic sugar since each of them has an equivalent "pure" UML representation. These equivalent representations, however, would make a diagram significantly more cluttered and difficult to understand.

The *merge operation* of this approach is very similar to Package Merge except that the operation takes place at the *class* instead of the package level. The *correspondence model* is not automatically created based on the names of classes but by manually using the equivalence operator which indicates the correspondence between two classes for their subsequent merge. This approach deals exclusively with UML class diagrams (metamodels) and is therefore even more specific than Package Merge which can be applied to all types of UML diagrams.

The authors mention that it is important that the composition leaves the original metamodels intact, so that they can still be used independently. Also, the newly composed metamodel should be capable of instantiating existing models created using the original metamodels (backwards compatibility).

An implementation of these operators has been added by the authors to the Generic modeling Environment (GME), a DSL design environment, developed in the Vanderbilt University.

4.2.1.4 Applied technique

In this work we have employed a similar variant of the merge techniques explained in this section with the following main differences:

- It allows duplicate information. This is because the tool we use does not impose this restriction;
- Its matching operation is empty, no correspondence model is needed. Therefore the combination can be performed automatically without user intervention;
- Conflict detection and resolution is not automated and has to be performed manually by the user after the combination has been completed.

For more details on our implementation see subsection 5.2.2.

4.2.2 Templates

In the context of model composition, templates are an *asymmetric* combination technique where one abstract *template* model, is instantiated by a *concrete* one via an *implement operation.* A template provides reusability by allowing the creation of general models that can later be further specified by *instantiating* pre-defined points called template *parameters.*

In Figure 4.2, a simple example of the template technique is shown. Two metamodels (that conform to the UML class diagram metametamodel) depict basic concepts of the same RPG application as in the previous examples. The template model *RPG_template* contains the Location, Character and Item classes while the extension model *RPG_tiles* specifies three new types of Tile: Obstacle, Trap and Door. The special feature in this technique is the symbol "|" before the Location class. This means that the Location class is a template parameter that has to be instantiated by a concrete element of another model (in this case a class since it is a homogeneous composition technique). In the *RPG_tiles* model, the Tile class instantiates or binds to the template parameter |Location.

4.2.2.1 Template Instantiation by Emerson and Sztipanovits

Template instantiation is an *asymmetric* composition technique where an abstract metamodel, the template, is *bound* to a concrete one. It is asymmetric because the template metamodel acts as the base model and is extended by the bound model. A *binding* relation specifies that a concrete entity (of the concrete metamodel) plays the role of an abstract entity (of the template metamodel). Abstract entities in a template are also called parameters since they, as in mathematical functions, can be replaced by a set of values or entities.

In [5], Emerson and Sztipanovits propose the use of template instantiation to overcome the limitations of techniques such as model merge and interfacing. The problem with these techniques, according to the authors, is that they are not well suited for the *multiple* reuse of metamodel fragments into the same composite metamodel. In other words, when performing a chain of consecutive compositions into one same metamodel, the weaving of a composition can be affected by the changes made when weaving a previous one.

Template instantiation, on the other hand, automatically creates new relationships between the pre-existing elements in a target metamodel with the template parameters of a common meta-modelling pattern. These common meta-modeling patterns are created

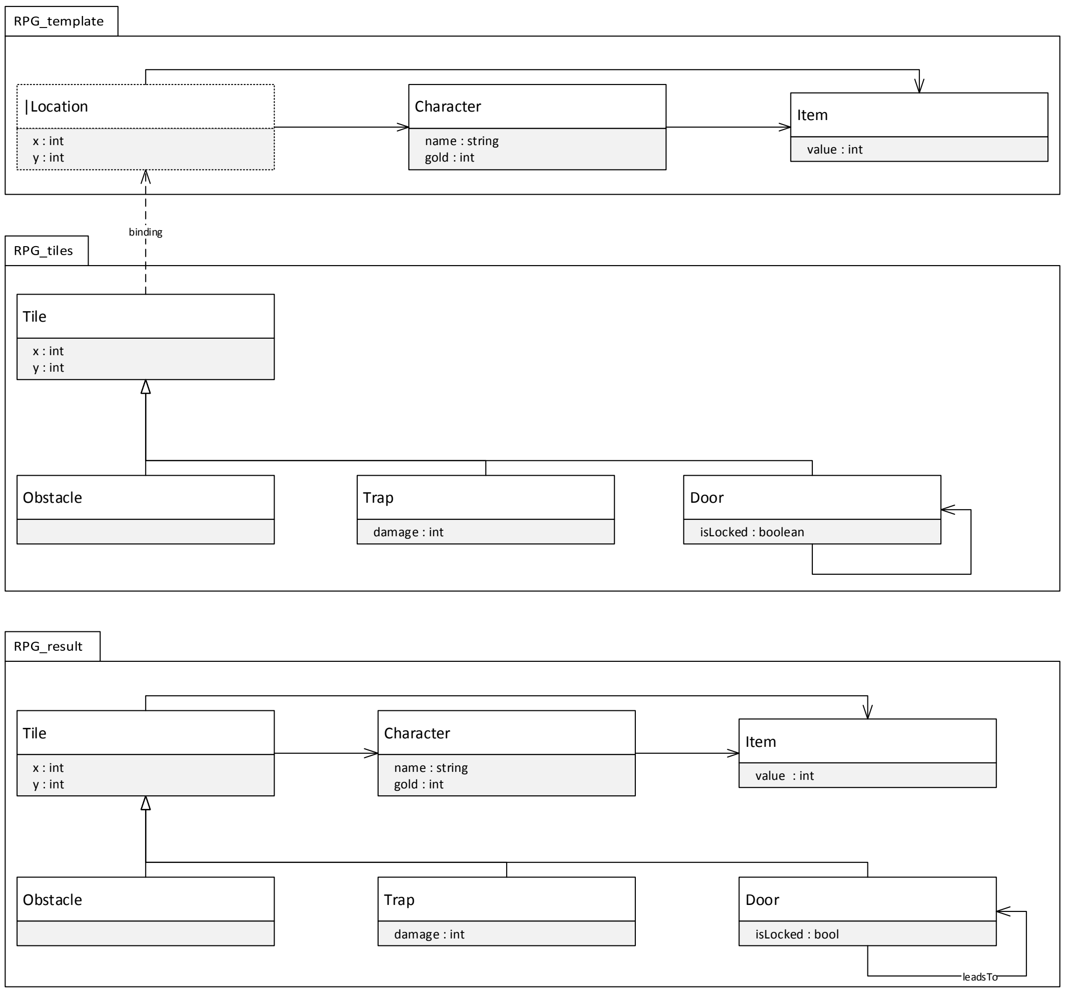


FIGURE 4.2: Example of the template instantiation technique in the context of the RPG DSL

from a set of templates of several metamodels that are commonly-occurring, e.g., State Charts, Data Flow graphs, Hierarchy, etc.

The authors provide a simple prototype for template instantiation in the GME metamodelling language which guides users through the selection of the template to be used and the assignment of the template parameters or roles to domain specific concepts. Then, it automatically edits the domain-specific metamodel in order to instantiate the template.

This composition technique can be applied to a broader spectrum of metamodels, the ones for which common metamodel patterns templates are provided and it is therefore heterogeneous. This means that it can be useful for abstract, concrete syntax as well as for the semantics of a language.

4.2.2.2 Templates in metaDepth

MetaDepth [23] is a textual meta-modelling environment tool that allows deep meta-modelling, in the sense that it supports an arbitrary number of meta-levels. This tool also has mechanisms that allow the definition of the three components of a language: abstract, concrete syntax and semantics. In [24], the mechanism of templates is used inside the metaDepth tool. This enables a more flexible definition of model and metamodel fragments, as they permit their connection.

They define a *metamodel* template as a metamodel where some elements (meta-classes, features, associations) are variables. The connection requirements for such variables are expressed through a *concept.* There are two types of concepts defined in the work of De Lara: structural and hybrid concepts.

Structural Concepts A *structural concept* is a specification of the structural requirements that need to be found in the abstract syntax of a model. Some elements of the concept can be variables or parameters, that have to be bound to concrete elements by another model. If a model provides parameters to a concept it is said to *instantiate* it since it satisfies the concept's requirements. This is useful since generic operations defined to work on a concept can then also be applied to any model that instantiates it.

Hybrid Concepts The binding that structural concepts require is a structural relation between the concept and a meta-model. This binding can be somehow limiting however, since it requires the concrete meta-model to have a similar structure as the concept in order to be possible. A more flexible mechanism is desired that allows binding concepts to a wider domain of meta-models with heterogeneous structures. *Hybrid concepts* provide this flexibility by hiding the specific structure of concepts behind appropriate operations. These operations form an *interface* (similar to Java interfaces) and impose less structural requirements to the bound meta-models. As a drawback, the meta-models are required to implement the operations specified in the concept.

Concept specialization Similar to class inheritance in Object-Oriented programming languages (OOPL's), *concept specialization* is a way to construct hierarchies of concepts where Liskov's substitution principle applies and therefore generic behavior defined for a concept is also applicable to all of its specializations (subclasses). Concept specialization is also a means to construct concepts incrementally.

In [8], Meyers et al. shows how to compose modelling languages by using these composition mechanisms. They perform composition on homogeneous metamodels due to the fact that all models have been defined with MetaDepth and thus conform to the same meta-model.

It is worth mentioning that this is one of the few approaches that composes all three components of a language, including the often neglected concrete syntax. The disadvantages of the approach are that it deals exclusively with textual languages defined in Meta-Depth (it is not a *generic* approach). Although it is possible to create an extra step in the process where a model defined in any modelling language is transformed into a model that conforms to the MetaDepth metamodel.

For future work, the authors of [24], mention they plan the addition of more advanced mechanisms for composition that allow bidirectional binding of two templates or a more flexible binding.

4.2.2.3 Applied technique

In this work we use a template instantiation technique similar to the one used in metaDepth where a metamodel template has some elements as variables. Concepts are not used in our technique and therefore there are no constraints for binding concrete elements to variable ones. This is how Emerson and Sztipanovits define the template instantiation technique where there is no mention of any constraints during the matching operation other than the basic typing and multiplicities.

4.2.3 Interfacing

Model interfacing is a technique that combines two models by defining an interface between them consisting of elements that do not strictly belong to either of them but allow them to interact with each other [5].

In terms of programming languages, an interface can be considered a method with an empty body that only defines its signature, *i.e.,* parameters and return types. The keyword *interface* is used in the Java programming language to define a class that only contains such methods and it is used as a contract that requires any subclass to implement them.

Adapting this idea to modelling and specifically to rule transformations, an interface may consist of abstract rules, i.e., without an implementation, a place-holder. A difference between abstract methods and rules is that the former impose a constraint or a contract on their implementation via their signature, while the latter do not have a signature and thus impose almost no constraints to the implementation. Rules can still be constrained by adding pre- and postconditions (expressed as rules as well) surrounding the abstract rule in question.

4.2.3.1 Applied technique

The interfacing technique used in this work is based on the hybrid concepts idea of section 4.2.2.2 where it is possible to create abstract operations, without an implementation or method body. These operations have to be implemented with information provided by the user through an *implement operation* (subsection 4.1.6). The operations follow the idea presented by Emerson in [5] since their implementation plays the role of the glue or elements between the two metamodels combined that do not strictly belong to any of them.

Chapter 5

DSL Combination in AToMPM

This chapter presents the framework that we have developed for combining domain-specific modelling languages in the AToMPM tool.

The proposed solution is built upon the work of Meyers et al [8] where they compose textual DSLs using the MetaDepth tool. It applies the techniques explained in chapter 4 to combine each of the three aspects of DSLs: abstract, concrete syntax and semantics. These three aspects are illustrated as models in the figures

5.1 Template DSLs

A *Template* DSL can be *extended* using an extension relation (see subsection 4.1.5) by a *Concrete* DSL resulting in a *Combined* DSL.

Both the Concrete and Combined DSLs are regular DSLs defined using the same meta-models for their syntax and semantics. Template DSLs, however, are different than regular DSLs in the following aspects:

- Their abstract syntax is defined using a variant of class diagrams where some classes can be of the special type *parameter* class. A parameter class is denoted by a & prefix in its name and it can be *instantiated* by a regular class of a different DSL. A class that instantiates a parameter can substitute it in any future occurrences, similarly to an inheritance relation. This technique is based on *templates* of subsection 4.2.2;
- Their operational semantics are defined using a variant of a transformation schedule language where some rule blocks can be marked as *abstract*, which means

that they have to be *implemented* in order for the operational semantics to be executable. This technique is based on *interfacing* of subsection 4.2.3.

Thus, a template DSL can be instantiated by instantiating its metamodel and implementing its required interface rules. After a template DSL has been instantiated, its semantics can be applied to the concrete DSL that instantiated it. This allows the semantics to be *generic*, since they can be used on any DSLs, as long as they instantiate the template DSL.

In the following subsections we explain the steps that have been taken to convert the secondary DSLs of chapter 3, Pathfinding and EventListener, to Template DSLs so that they can be used in the combination framework that we have created. The RPG DSL is not converted to a Template since it plays the role of the Concrete DSL in the combination phase discussed later.

5.1.1 Pathfinding Template

The Pathfinding DSL, presented in section 3.2, is used to create graphs with a source and a destination node and to then calculate the shortest path between them.

To convert the Pathfinding DSL into a template their abstract syntax and semantics models were modified as follows: First in the class diagram metamodel, the node and edge classes are converted into parameter classes by adding a & prefix to their names. The fields contained in these classes can also be converted to parameter fields, which behave in an analogous way as parameter classes: they have to be instantiated by a concrete field. But for the case of this work this is not used and therefore not done, although having parameter fields would make the template even more generic and flexible. The final form of the template DSL can be seen in Figure 5.1;

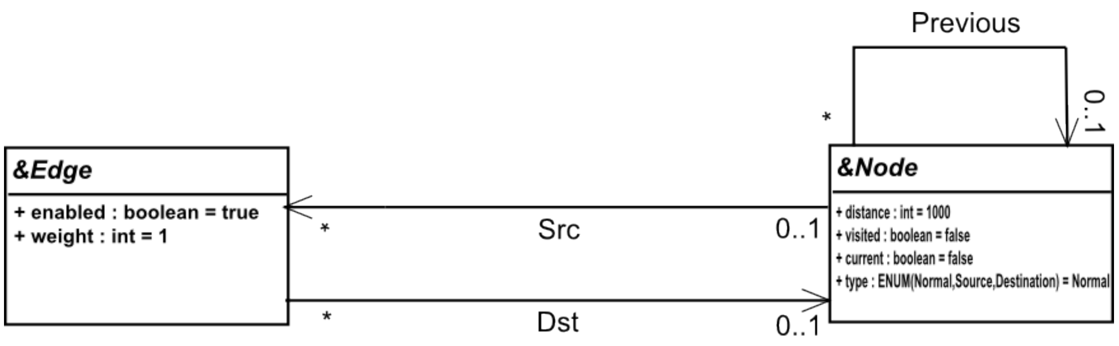

Previous
*
0..1
&Edge
+ enabled : boolean = true
+ weight : int = 1
*
Src
0..1
*
Dst
0..1
&Node
+ distance : int = 1000
+ visited : boolean = false
+ current : boolean = false
+ type : ENUM(Normal,Source,Destination) = Normal

FIGURE 5.1: The abstract syntax *SimpleClassDiagram* model of the Pathfinding template

For the operational semantics, the following rules are made *abstract*:

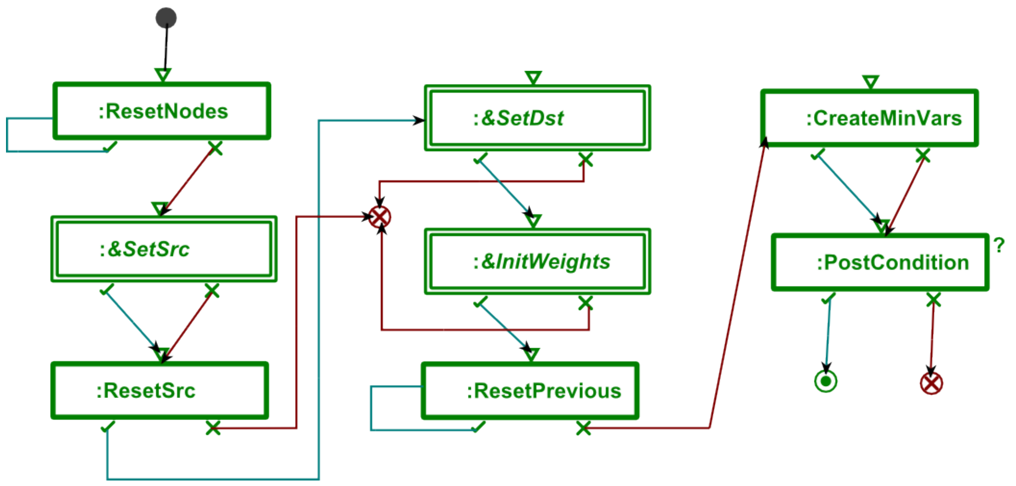


FIGURE 5.2: The sub-transformation used in the Pathfinding DSL, abstract rules are denoted by a & prefix in their name.

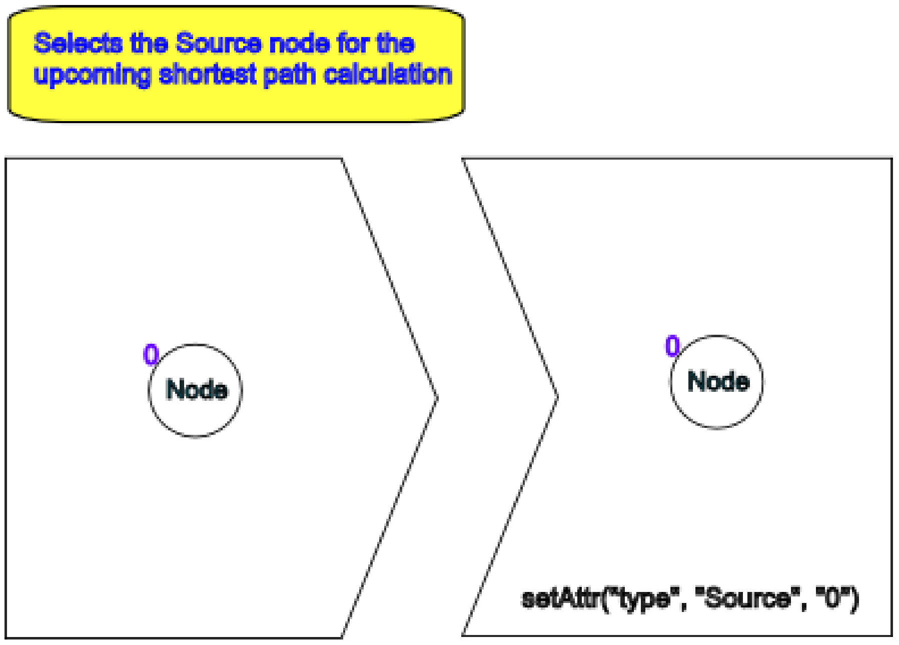

Selects the Source node for the upcoming shortest path calculation
0
Node
0
Node
setAttr("type", "Source", "0")

FIGURE 5.3: The default implementation of the `&SetSrc` abstract rule of the transformation shown in Figure 5.2. The text in the yellow area is a description of the expected behaviour of the abstract rule (it adds no extra behaviour).

- *setSrc* and *setDst*, that marks a node as the source and destination node, respectively, before the start of a new shortest path calculation;
- *initWeights*, which sets all the initial weights for all the edges in a graph.

All of these rules take place during the *T_initNodes* sub-transformation, which is executed at the start of the main transformation. To make a rule abstract, their name is extended with the prefix &, similarly to parameter classes. In addition, a new rule model is created for it and placed in the *Interface* directory of the DSL. This rule model can be empty, or have an implementation that describes the default behaviour of the rule, for example the *setSrc* rule's default behaviour might be to mark a random node as the source.

An example of a DSL that could instantiate this template could be a network of cities connected by highways. The cities would be the nodes, the highways the edges, and the distance between the cities could represent the weight of the edges. Another example is a computer network where nodes pass data through network links (edges). The delay for the data to traverse each link can be represented by the weight of an edge. Afterwards, for both examples, the shortest path between two nodes (network nodes or cities) can be calculated using the Pathfinding DSL's algorithm.

5.1.2 EventListener Template

The Event Listener DSL, presented in section 3.3, applies the Observer pattern for key-presses event handling. By converting it into a template, other languages can benefit from event handling capabilities.

In order to transform this DSL into a Template, the *KeyListener* class becomes a parameter class. This allows a class from a Concrete DSL to instantiate *KeyListener* and be able to register events of the type *KeyEvent*. The abstract syntax, modelled using class diagrams, can be seen in Figure 5.4.

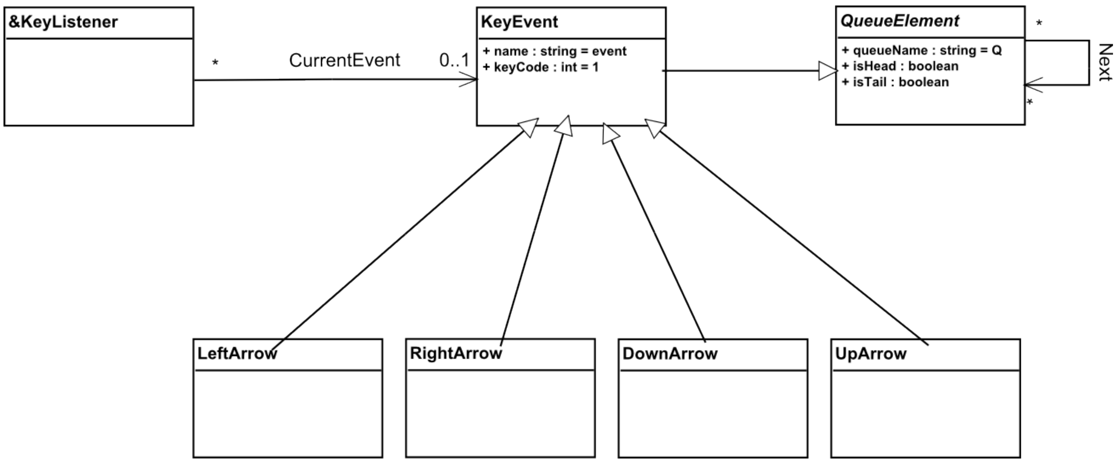


FIGURE 5.4: The abstract syntax *SimpleClassDiagram* model of the EventListener Template DSL

For the semantics of the DSL, the variable elements are the action rules, i.e., *LeftAction*, *RightAction*, etc. These four rules are made abstract by adding the & prefix to their names and by moving them into the Interface folder. In these rules, the user of the template must manually specify the consequences of pressing one of the four supported arrow keys.

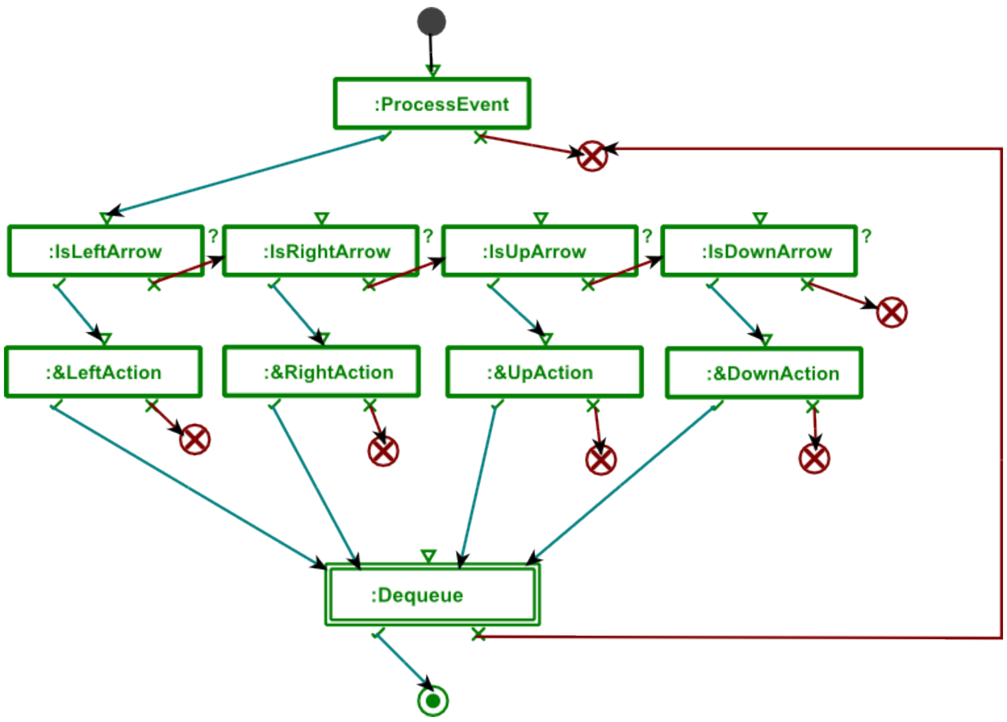


FIGURE 5.5: The operational semantics of the EventListener Template DSL, the abstract rules are denoted by a & prefix in their name.

This DSL has been created with the idea of combining it with the RPG DSL, so that pressing an arrow moves a character of the game in the corresponding direction, but it can be easily extended to support more keys for many other functions.

5.2 Combination Process

In this section we walk through the activities involved in the combination process from the user's perspective. The flow of activities that take place during the combination are displayed in Figure 5.6. This figure depicts the activity with a model of the FTG-PM language [25]. It consists of two sub-diagrams: on the left side, the Formalism Transformation Graph (FTG) is shown, which declares the set of languages involved in the process as well as the transformations between those languages; on the right side, the Process Model (PM) diagram is depicted, which describes the control and data flow of activities.

The inputs for the combination process are a Template DSL, explained in section 5.1, and a Concrete DSL, a regular DSL created by the user that is to be extended by the template. Each DSL (Template or Concrete) consists of the following three parts, that are represented as a single or multiple models:

- Abstract syntax metamodel, defined using the `SimpleClassDiagrams` language;

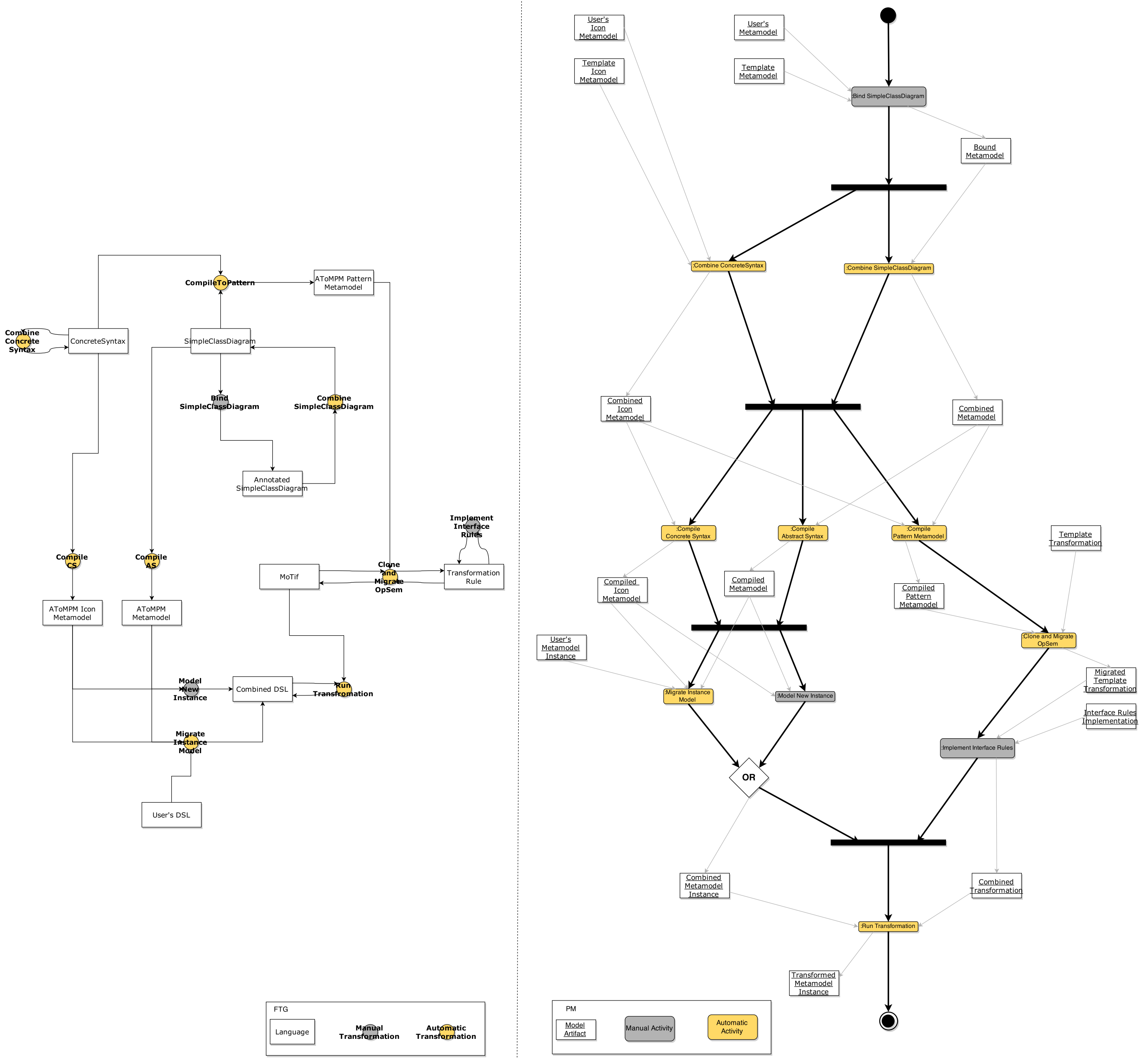


Figure 5.6: FTGPM of the combination process from the point of view of the user

- Concrete syntax metamodel, created using the `ConcreteSyntax` language;
- Semantics model, defined using the `Transformation` (for schedules) and `TransformationRule` (for rules) languages. The semantics model is usually composed of one or more transformation schedules where each can contain at least one transformation rule.

The process starts with the activity `:Bind SimpleClassDiagram`, which transforms the template and user's metamodels into a bound metamodel, an extension of class diagrams that allows the creation of binding links between classes, associations and fields. This activity is coloured grey to denote that it is not automated and needs the intervention of the user, who has to specify the binding relations between the metamodels. An example of a bound metamodel is shown in Figure 5.8. The FTG diagram at the left hand side classifies the bound metamodel as an instance of the `Annotated SimpleClassDiagram` language since it still conforms to the `SimpleClassDiagram` language but is annotated by additional elements (the binding links) that belong to the `TemplateCombination` toolbar. This toolbar provides the user with the necessary tools for specifying the links that are involved in our DSL combination technique.

The next two activities `:Combine Icon Metamodels` and `:Combine SimpleClassDiagram` can be done in any order and are executed automatically by our implemented framework in AToMPM. The `:Combine Icon Metamodels` applies the technique described in subsection 5.2.2 to produce an Icon Metamodel for the combined DSL. The `:Combine Icon Metamodels` activity performs a model transformation that converts the annotated bound metamodel into the combined metamodel by replacing binding links according to the technique of subsection 5.2.1. An example of a combined metamodel after performing this activity is illustrated in Figure 5.8.

After having combined the abstract and concrete syntax models of our participant DSLs, they have to be compiled into a language. In AToMPM, a language can be created by compiling these two elements, a specification of its semantics is not necessary for this purpose. The compilation occurs in the three `:Compile` activities shown in the PM diagram. The third activity compiles the abstract syntax into a pattern metamodel. A pattern metamodel is a *RAMified* [16] version of the abstract syntax metamodel that is used in the creation of transformation rules.

With the compiled pattern metamodel at hand, the template transformation (specifying its operational semantics) can be migrated to the new formalism. This is performed by the `:Clone And Migrate Semantics` activity, which first copies the directory `OpSem` of the template formalism into that of the combined formalism. Then, it executes a retyping operation on all the models in the cloned directory and migrates their pattern metamodel to the pattern metamodel produced by `:Compile Pattern Metamodel`. The result of this activity is the `Migrated Template Transformation` artifact, that comprises all the schedule and rule models of the template operational semantics. This migrated transformation is now defined in terms of the newly created types of the combined language and can therefore be applied to any instance that conforms to it.

The migrated transformation is not completely ready for usage though, first any abstract or interface rules that it contains have to be implemented by the user. This is done via the `:Implement Interface Rules` manual activity, detailed in subsection 5.2.3. It takes the implemented rules as input and produces the final combined transformation artifact that is ready to be applied to an instance of the new combined language.

To create an instance of the new language, the user has two possibilities: creating a new instance from scratch, using the compiled concrete and abstract syntax metamodels to do so; or it is also possible to migrate an existing instance model of either the template or the user's language. The `:Migrate Instance Model` applies a similar mechanism as the rule migration activity mentioned previously. In any case, the result of any of these two activities is an instance of the newly created combined language.

Finally, the diagram includes the `:Run Transformation` activity, which is not technically a part of the combination process but it is the goal that it tries to accomplish. This activity employs the transformation engine present in AToMPM by loading the combined instance model into the canvas and executing the combined transformation, producing a variant instance model that still conforms to the combined language.

The following subsections, explain in more detail the combination of each language component, i.e., abstract syntax, concrete syntax and semantics, that take place during the combination process described above.

5.2.1 Abstract Syntax

The abstract syntax of the two languages are combined using templates, where one of the participant abstract syntax metamodel plays the role of the template and the other that of the instantiating metamodel. Templates are used in the Template Instantiation technique [5] explained in subsubsection 4.2.2.1 as well as in De Lara's MetaDepth implementation in [24].

Our approach does not include concepts so it does not demand any structural requirements from the instantiating metamodel. It is the responsibility of the user template to use a metamodel with a compatible structure as the template. However, it does require the implementation of some elements of its operational semantics, in a similar way as hybrid concepts [24], this is discussed in subsection 5.2.3.

The template is bound by tracing `binding` links between the entities of the template and the user's metamodels. Since the template is designed for combining metamodels defined in the class diagram language, three kind of bindings can be performed, depending on the type of entities involved: class, association and field bindings. In the next sections we describe how each of these types of entities are bound and combined.

5.2.1.1 Binding Classes

The binding of a class is performed by the user by tracing a `binding` link between a parameter class (denoted by a & prefix) of the template metamodel and a class of the user metamodel.

There are different approaches for realizing a binding relation between classes but the one we deemed more useful was inheritance, as explained below. An alternative approach is replacing all instances of the parameter class by an equivalent instance (with an identical state) of the bound class. This can be done in AToMPM by modifying the instance models raw files directly (in a similar way as model migration is done in subsection 5.2.3). If the class that is bound to a parameter becomes a subclass of that parameter, it means that it can take the place of any instances of the parameter class (Liskov's substitutability principle). This coincides with the semantics of a binding, where an abstract parameter is replaced by the variable or value that instantiates it.

Using inheritance to realize the binding has the additional benefit of allowing the reuse of transformations by using the option `match_subtypes` in their transformation rules. By activating this feature, transformations that were defined to match parameter classes will also match any subtypes of that class, in this case the class of the user's metamodel that is bound to it. This allows the creation of *generic* transformations defined in terms of a template (and its parameters).

5.2.1.2 Binding Associations

For associations it is not possible to use the same approach as for classes since AToMPM does not support association inheritance. The chosen solution is to represent associations as classes, at least the associations that are template parameters and are going to be usually sub-classed. This allows association bindings to behave in the same way as class bindings, the downside of this approach is that it requires the metamodel to be modified.

An alternative approach is to simulate the inheritance relation by storing the information of the super and sub-associations and then, using this information, modifying all instance models by replacing all occurrences of the super class by ones of the sub-class. The replacement could be done using a simple model transformation and it would be similar to a model migration operation, but it would have to be executed each time that the instance model changed since a new instance of the association could be added.

Parameter associations are denoted in the same way as classes, with a & preceding their name and are bound by connecting it via a `binding` link to an association of the user metamodel.

5.2.1.3 Binding Fields

To bind two fields, the user has to trace a link connecting the two classes where these fields are contained and then select the *FieldBinding* link type. Afterwards, the user must fill in the names of the source and destination fields. The source field (i.e., the template field) is denoted by a & prefix in its name.

In order for two fields to be bound successfully to each other, the following requirements have to be met:

- The class where the link originates has to be a template class, which are denoted by a & prefix in their name;
- The two classes that contain the fields in question must be bound to each other by a *Binding* link that originates from the template class;
- The two fields must be of the same type, either primitive or object type.

If any of this requirements is not met, the transformation process will not recognize the field binding and skip it.

As mentioned before, we use the `SimpleClassDiagram` formalism to model the abstract syntax of domain-specific languages. This formalism does not explicitly model the field objects of a class, there are no field objects available as it is the case for class and association objects. Instead, the fields are encoded in a dictionary or map inside their parent class object. Therefore, the binding of fields can not be realized by inheritance as in the previous sections. The alternative solution that we use is to store the binding information (i.e., the two fields that are being bound to each other) within the class where the binding originates, also referred to as the *template* class.

To store the binding information we the AToMPM feature of hidden attributes. An attribute with a $ prefix in its name becomes a hidden attribute, which means that it is not visible at the instance level (when an instance of the class in question is created)

The user of the language is not aware of any hidden attributes since they are not shown as part of the entity in AToMPM's object editor.

To access the hidden attribute, the template designer has to use AToMPM's meta-modelling API within transformation rule's action and condition code. To read or modify the hidden attribute, the functions `getAttr()` and `setAttr()` can be used respectively. These functions take as first parameter the name of the field (including the $ symbol).

The information of a field binding is stored automatically by the template combination transformation.

Since the intention of the parameter field is for it to be a place-holder, its value is never used.

So this field can be used for storing the binding information, namely the name of the user's field.

The name of the field can be retrieved by using the meta-modelling API functions from within rules in the following way:

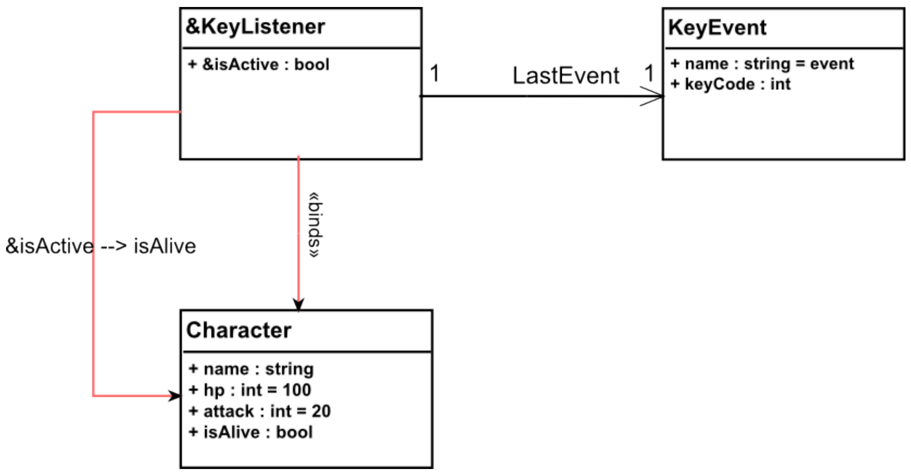
```
# get the name of the bound attribute stored in the hidden attribute
attrName = getAttr('$&paramAttr')
# Next, get the value of the attribute using the obtained name
attrValue = getAttr(attrName)
```

In Figure 5.7, an example of a field binding can be seen. Figure 5.7a shows the binding configuration, where the template field *&isActive* is bound to *isAlive*. Putting this in the context of the DSLs, the *Character* object, which now also behaves as a *KeyListener*, will only be active (i.e., listen to incoming events from the environment) as long as the character is alive (its hp larger than zero) in the game. Figure 5.7b shows the class diagram after the binding link has been processed. The information is now stored in the hidden attribute $&*isActive*. Its value is now of type string and it contains the name of the other field that participated in the binding, i.e., *isAlive*.

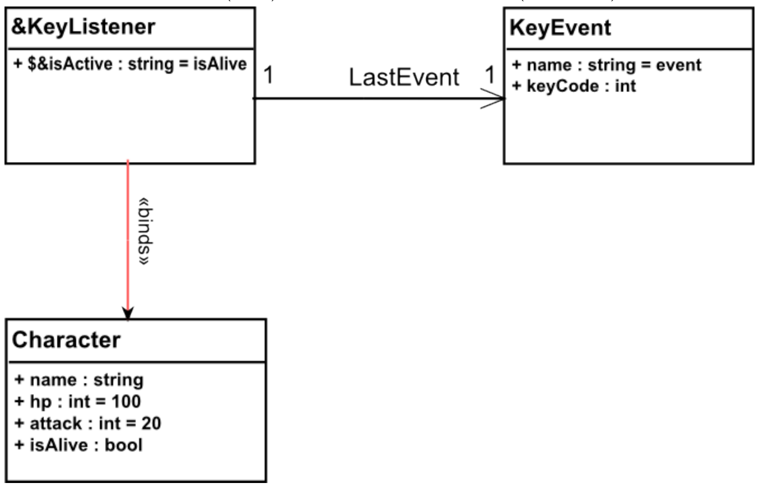
sec:combine-concrete-syntax

5.2.2 Concrete Syntax

The combination of concrete syntaxes is performed by applying a variant of the meta-model merge technique explained in subsection 4.2.1. This operation combines two models defined in the `ConcreteSyntax` language, the default way of defining concrete



(A) Example of a field binding between classes of the EventListener Template DSL (top) and the RPG DSL (bottom)



(B) Example of a field binding between classes of the EventListener Template DSL (top) and the RPG DSL (bottom)

FIGURE 5.7: Example of a field binding operation

syntax in AToMPM. Because of the nature of the models being combined, one being a template with abstract parameters and the other a concrete one, it suffices to perform a full union of the two concrete syntax definitions to create a new icon definition that corresponds to the combined abstract syntax metamodel. This can be done by loading one model into the canvas, inserting the other model and saving it as the new combined icon metamodel. The user can modify the result if he desires as long as the correctness of the icon metamodel is preserved.

In case elements are found to have the same name, they are not merged but both added to the resultant icon metamodel. Name clashes can be ignored because AToMPM does not throw an error in case of duplicate icon names, it simply uses the one that was created first (the one with smallest ID). Conceptual equivalences, i.e., two entities with different names that represent the same concept, are difficult to detect automatically,

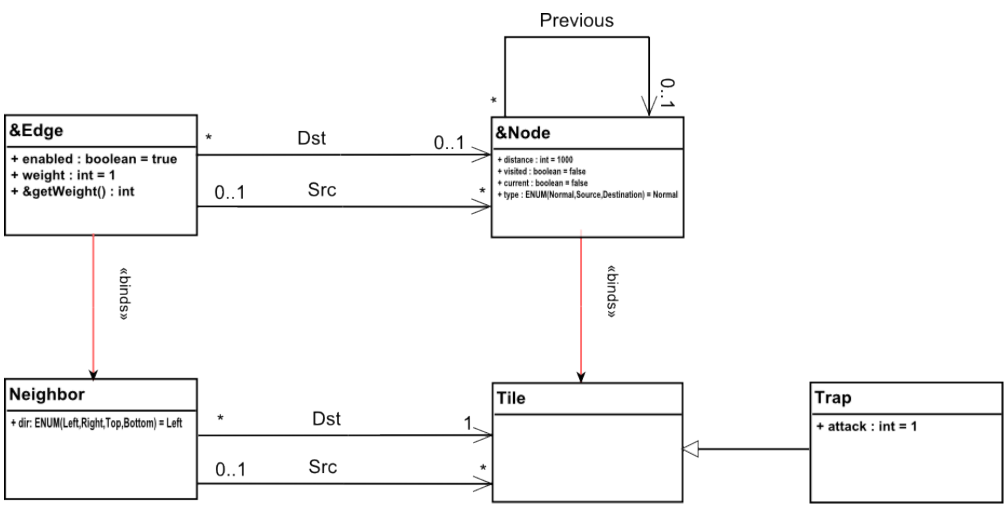

Previous
*
0..1
&Edge
+ enabled : boolean = true
+ weight : int = 1
+ &getWeight() : int
*
Dst
0..1
0..1
Src
*
&Node
+ distance : int = 1000
+ visited : boolean = false
+ current : boolean = false
+ type : ENUM(Normal,Source,Destination) = Normal
«binds»
«binds»
Neighbor
+ dir: ENUM(Left,Right,Top,Bottom) = Left
*
Dst
1
0..1
Src
*
Tile
Trap
+ attack : int = 1

FIGURE 5.8: Binding of the `PathfindingTemplate` and the `RPG` languages

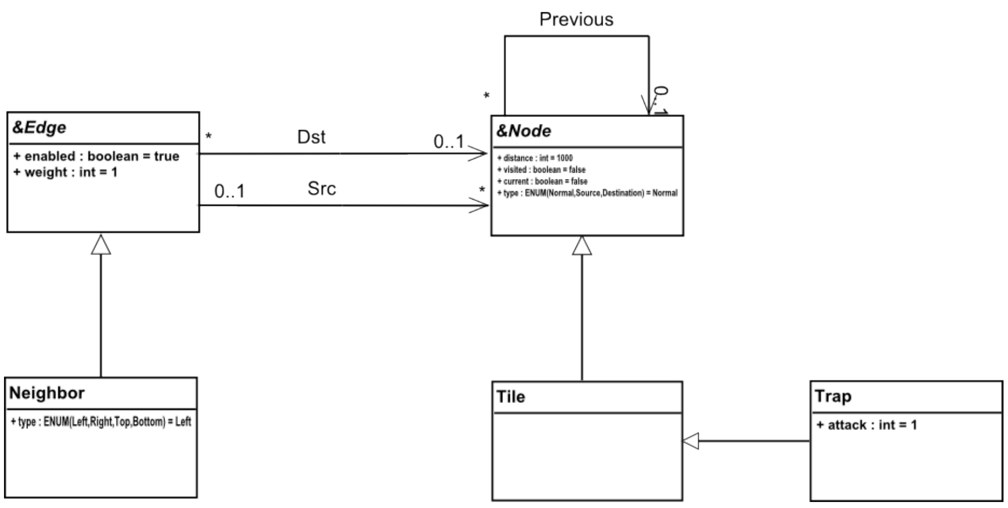

Previous
*
0..1
&Edge
+ enabled : boolean = true
+ weight : int = 1
*
Dst
0..1
0..1
Src
*
&Node
+ distance : int = 1000
+ visited : boolean = false
+ current : boolean = false
+ type : ENUM(Normal,Source,Destination) = Normal
Neighbor
+ type : ENUM(Left,Right,Top,Bottom) = Left
Tile
Trap
+ attack : int = 1

FIGURE 5.9: The produced abstract syntax model after processing the binding relations of Figure 5.8

and are therefore left for the template user to realize if he so desires.

5.2.3 Semantics

For designing the semantics of a DSL we also use DSLs, namely:

- `TransformationRule`, a formalism provided by the AToMPM designers that allows the creation of graph transformation rules and;
- `MoTif`, for specifying the control flow or also called *scheduling* of the transformation rules created with the previous formalism. This formalism is also included by default with the latest version of AToMPM and has been designed by Syriani [26].

The combination of the semantics of two languages is performed on schedule granularity. Rules are kept separate, they cannot be *merged*. Therefore the combination consists in specifying an interleaving of the rules contained in the two to-combine schedules. This type of combination is non-intrusive at the rule level since it does not modify the original rules and intrusive at the schedule level. Additionally, new rules may have to be created (by the user) in order to act as an interface or *glue* between the rules of the two combined schedules.

We consider two approaches for combining the semantics of two DSLs, they are both similar in that they require the user to implement a set of required behaviours at the moment of binding. They differ in the way that these behaviours have to be specified.

5.2.3.1 Abstract Rules

The first approach is to define *abstract rules* in the schedule of the template language. These rules have no implementation at design time, but do require and expect a certain behaviour (specified by their name and a small description) at run (simulation) time. They are similar to the concept of abstract methods or functions used in programming languages (e.g., Java or C#), that contain no body and have to be overridden by a concrete method of a derived class.

To represent them in the scheduling language that we are using (MoTif), we employ the *composite rule* construct. MoTif's composite rules contain a reference to another (sub-) transformation instead of a rule as regular rules do. Additionally, we mark abstract rules with a & prefix in their name to differentiate them from regular composite rules and also delineate them with an orange outline. This is a purely syntactic change and has no effect to the scheduling language (they remain being composite rules). It is intended so that the user of the template knows which rules have to be implemented in order to instantiate said template.

The procedure of implementing these rules by the user can be eased by the template designer if he creates basic rule models (with very simple implementation) and links them to the transformation scheduler using the `path` attribute, that points to the location where the model file is located in the AToMPM directory. Creating these models beforehand allows them to act as default implementations in case the user decides not to provide one. They are also useful as an example for the user on a possible way of implementing these abstract rules. An example of the default implementation of an abstract rule is shown in Figure 5.3. It can also be seen that it is accompanied by a brief description of the expected behaviour for the rule, this provides further assistance to the user of the template.

In Figure 5.2, a fragment of the `PathfindingTemplate` DSL's transformation scheduler is shown. It contains three abstract rules (denoted by a & prefix in their name) that are used for initializing purposes of the transformation.

5.2.3.2 Operations

The second approach is inspired by De Lara's hybrid concepts [24].

In order to adapt the functionality of operations as used in hybrid concepts, the invocation of operations from within rule condition or action code needs to be supported. Operations should be called in a similar way as the Transformation API functions `getAttr()` and `setAttr()`, that take the label of a pattern element as first parameter. The difference between these two is that the API functions are defined beforehand (in the source files of AToMPM), while operations are to be implemented during the instantiation of a template.

It is worth noting that operations called from a rule's LHS pattern have to respect the constraints of an LHS, namely that they can not make any modifications or have side effects to the host graph, in a similar way as the `getAttr()` function. If an operation does have side effects, it must not be allowed to be invoked from within the LHS condition code, in the same manner as the function `setAttr()` is not. While with the example functions `getAttr()` and `setAttr()` it is clear for AToMPM which one has side effects and which does not, with operations this is not trivially detectable by AToMPM. To detect this, a transitive search could be performed in order to determine if an operation is contained in the RHS or LHS blocks of a rule. The solution we have found is to include this information in the definition of an operation: whether or not it has side effects. This can be done with an additional attribute in AToMPM that the user implementing the operation has to fill in at the moment when it instantiates said rule.

In this section we consider two approaches for implementing operations in AToMPM: a modelling and a coded approach.

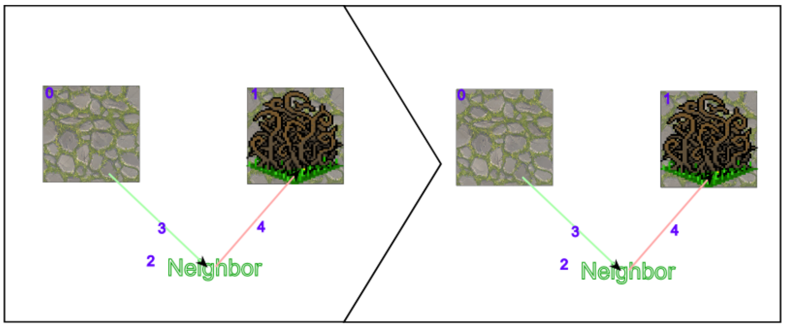
Modeling approach This approach for implementing operations in AToMPM is the most suitable to the context of domain specific languages, where models take a central role. It consists in defining operations using transformation rules, widely used throughout this work to specify behaviour. In order to support this integration, two functionalities are required from transformation rules: passing data between them during an execution of a transformation and the ability to nest one rule (or a transformation of multiple rules) inside the condition or action code of another. This approach proposes the creation of a new type of rule called *operation rules*.

Passing data Exchange of information between transformation rules is necessary for allowing rules to take input and produce output data. The existing transformation framework of AToMPM offers two ways of accomplishing this, pivots and session keys:

Pivots allow rules that are executed consecutively to share one or more common pattern elements. This can be used to speed up the matching process by making an element that is often matched across different rules a pivot in one rule and pass it to the next one and so on, in this way avoiding extra work on the subsequent rules. But pivots can also be applied to obtain the desired functionality of passing input and output values between rules. Note that only metamodel elements can be passed as pivots, so if an operation's return value is of a primitive type such as Integer or Boolean this would have to somehow be contained in a metamodel element. This could be achieved by having an object in the canvas with the only purpose of being the pivot and holding input and output values during the execution of a transformation. In order to support all possible types of values, this pivot object would need to either have a field for each possible type or store the value as a string and later parse it.

A simpler way of obtaining the same functionality is using session keys instead of pivots. Session keys are dictionary entries stored in a *transformation session*. A transformation session enables user data to be easily accessed and stored across several rules and transformation executions. It is only cleared when a transformation is (re)-loaded in the AToMPM client. The methods `session_get(key)` and `session_put(key, val)` respectively enable retrieving and setting or updating a stored value. Operation rules can set a session key with an appropriate name e.g., `<NameOfRule>_OUT` with the returned value in the action code of their RHS. Subsequently, this value can be retrieved by rules with the `session_get(key)` function from their action and condition code. Session keys can replace the functionality of pivots by storing the id of a specific object and subsequently in another rule matching objects against that id. However, since pivots are implemented in the source code of AToMPM they might be more efficient, but they are not as suitable as explained above. A combination of using pivots (for elements) and session keys (for primitive data) seems like the most ideal solution.

Since the underlying implementation of the session keys is done in Python, which has dynamic typing, the values stored can be of any type, including a data structure such as a dictionary where multiple values could be stored in one entry. This allows the specification of operations with multiple return values. A possible extension to simplify the usage of session keys is to encapsulate them into new functions that are specific to retrieving input and producing output values. For example, new API functions such as `returnOutput(val)` and `getParameters()` could automatically use the name of the



```
params = session_get("getWeight_IN")
# read the nodeId parameter
nodeId = params[0]
result = getAttr("$atompmId", "2") == nodeId
```

```
# Edges to a trap should have a higher weight
toTrapWeight = 10
session_put("getWeight_OUT", toTrapWeight)
```

FIGURE 5.10: Transformation rule of the `getWeight()` operation using session keys. The condition and action code of each pattern block can be seen below it.

rule where they are contained as the key. This has the advantage of reducing possible errors while handling session keys as well as making it simpler to design operation rules.

In Figure 5.10 the operation rule `getWeight()` used in our running example can be seen. It uses session keys to pass and receive parameters, it takes the ID (`$atompmID`) of the caller `Neighbor` link as input and it returns its appropriate weight depending on the tiles to which it is connected. This rule takes part of the binding process to combine the Pathfinding and RPG languages of our running example.

Nested rules The second functionality needed to support operations is to allow them to be called from within rule condition and action code, similarly to how the functions `getAttr()` and `setAttr()` can be called.

This can be achieved by defining an operation in one or several rules and include them in a transformation schedule. The schedule can then be executed with a function from within pattern code of an ongoing rule execution. Input parameters would have to be stored in session keys as explained before executing the nested transformation and, after its execution, the result would be retrieved from another appropriate session key. Unfortunately, the current version of AToMPM does not support the loading and running a transformation from within a rule's code. It is possible however to modify AToMPM's source code and add this functionality, but this is beyond the scope of this work.

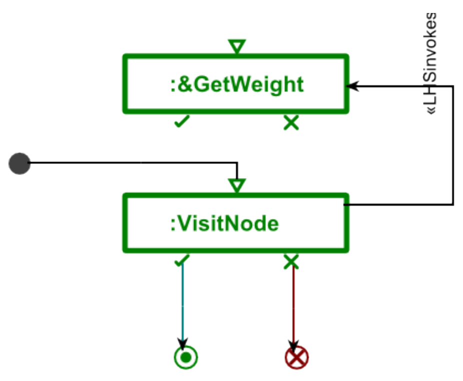
As mentioned earlier, it is important that an operation that is called from an LHS block has no side effects. An alternative option to achieve this is to simulate the behaviour of nested rules with the available tools at our disposal, i.e., transformation rules and schedules. There are two cases in which operation rules can be invoked that we have to consider.

One occurs when an operation is invoked from within an LHS pattern. In this case the execution of the rule has to be paused right after a pattern has been matched and before the condition code is executed, since this code uses the return value of the operation. At this moment the control is given to the invoked operation until it has finished. Any needed parameters are passed to the operation rule using session keys. When the operation completes, the control is given back to the rule along with the output of the operation, which is used in the condition code. If the condition of the rule fails, the matching engine has to backtrack to a different match. In that case the operation will have to be executed again before executing the condition code. This is repeated for as many times as the condition code is re-entered, since, for every match candidate, the operation might have a different input and produce a different result.

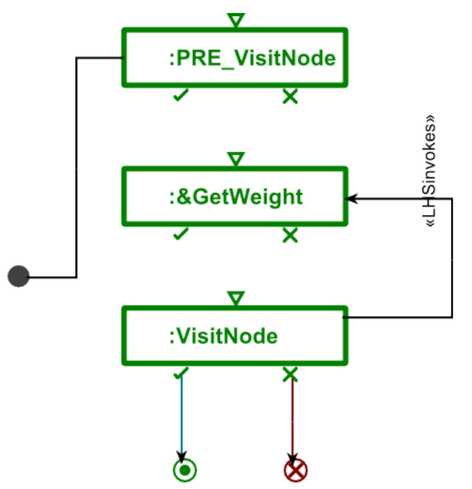
An attempted approach for simulating the invocation of operations from within LHS blocks is adding an extra rule to the schedule called *pre-rule* that precedes any rule that has such an invocation. This pre-rule is used to match the same elements as the rule that invokes the operation, by having an identical LHS pattern. However, the pre-rule does not perform any changes to the match, its RHS contains the same elements as its LHS — thus leaving the match exactly as it was found. Upon completion, the pre-rule stores the necessary match data, such as the `atompmId` of the matched elements, in order to allow the second part of the rule to continue its execution.

A step-by-step example of this solution is shown in Figure 5.11. Figure 5.11a shows the initial setup of the schedule, the `LHSinvokes` link can be added to indicate that a rule invokes an operation from its LHS pattern. Note that this is not a valid transformation specification and thus has to be transformed into one. In Figure 5.11b, the pre-rule is created and all incoming links to the rule (in this case only an `initial` link) are rerouted to the pre-rule. Then, in Figure 5.11c, the outgoing failure link from the rule is instead changed to originate from the pre-rule. If the pre-rule, that takes care of the matching phase, fails to find a match, the entire rule (now consisting of the pre-rule, operation and rule) would also fail.

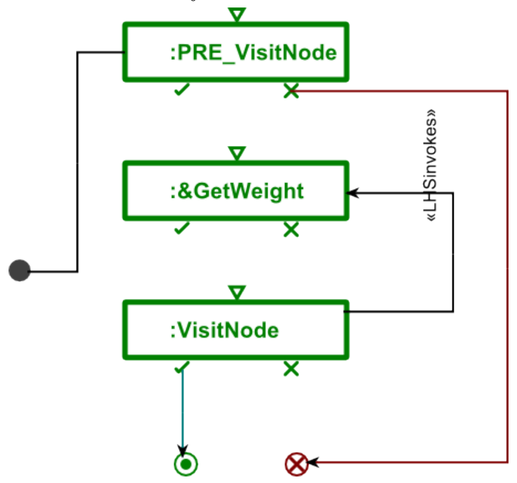
Finally, in Figure 5.11d, the internal links between the pre-rule, operation and rule are created. If the pre-rule succeeds in finding a match, the operation (`&GetWeight`) is executed. The operation should always succeed in finding a match since it retrieves the match from the pre-rule and reapplies it, for this reason it does not have an outgoing `fail` link. In case it fails the intention is for the transformation controller to throw an error. After the operation succeeds, the actual rule is executed; this rule makes use of the value produced by the operation in its condition code to find an appropriate match, in the shown example the weight of the node has to meet a certain criteria. It is possible that the rule condition fails, because the weight of the node was not appropriate for



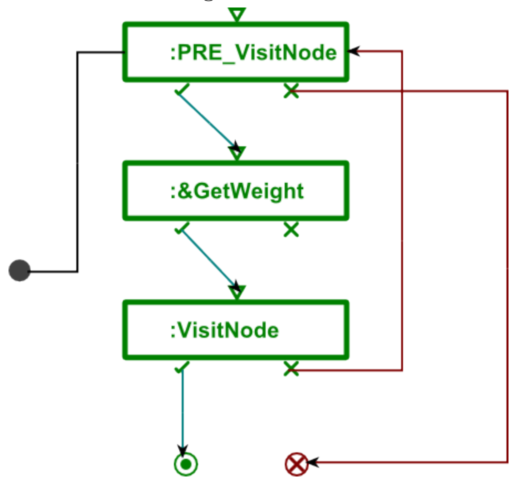
(A) An operation is invoked from a rule, denoted by the `LHSinvokes` link



(B) Pre-rule creation and rerouting of incoming `initial` link



(C) Rerouting of outgoing `failure` link to originate from the pre-rule



(D) Final form of the schedule after drawing of internal links

FIGURE 5.11: Transforming a fragment of a schedule to support an operation rule

example; then the pre-rule has to be executed again and therefore the failure link is added from the rule to the pre-rule. In order to avoid an infinite loop if no match is found, the rule and pre-rule need to keep track of a list of nodes that have been considered in the pre-rule. This list, called the `matched` list is stored using session keys so that it is accessible to both of them.

These steps have been modelled using a higher-order transformation schedule and is thus fully automated in AToMPM.

If a rule invokes the operation from within its RHS pattern's action code, no modifications to the schedule have to be made (like in the case where invocation occurs in the LHS), since if an operation is called from an RHS, it means that the rule has already been applied successfully. Having the operation rule scheduled after the rule that invokes it would then suffice to simulate the behaviour.

This approach has several disadvantages compared to the standard way of rule matching. First, modelling (part of) a matching algorithm using transformation schedules and rules is much more complicated than implementing it in a programming language. This is needed because of the limitation of not being able to call nested transformations from within LHS's or RHS's code. Second, the access to the internal matching data is limited from the modeler's perspective and thus a parallel set of variables have to be created and kept track of. Lastly, each matching would take extra work and thus it takes much longer for the transformation to finish.

Coded approach An alternative that is much more viable with the current framework of AToMPM is to define operations using code instead of transformation rules. Since the code is to be executed from within conditions and actions, it has the same properties, i.e., it is implemented using either Python or Javascript and it has access to the Transformation API functions of AToMPM normally accessible from pattern code.

The implementation of this approach consists of adding the snippet where a given operation is defined to the condition or action code blocks that invoke them. This can be performed using a mix of transformations and AToMPM scripts, by loading the models of the appropriate rules, modifying their LHS or RHS blocks accordingly and saving the model again. Or it can also be done by modifying the source code of the transformation rule models files directly. Note that it can not be done beforehand by the template designer since the snipper with the implementation does not exist at that time, it has to be implemented by the user of the template (the one that instantiates it).

5.2.4 Model Migration

This section briefly discusses the model migration technique applied in the combination process. In order to use the operational semantics of the template DSL on the newly created combined DSL, the models that specify the semantics have to be migrated to the new formalism. This migration consists in replacing all the type references of one metamodel to another one. It has to be applied to all the rule models that reference the pattern metamodel of the original template formalism. In addition, transformation models that have references to these rules have to be changed as well so that they refer to the cloned set of rules instead. These references are strings that contain the path (relative to the root of AToMPM's folder) to the file where such rule is defined. Furthermore, instance models can also be migrated as well, although this is not essential to the combination process as it is migrating the operational semantics.

Model migration allows for an interesting choice between a key question mentioned by Vallecillo in [21]: whether users should know about the combined language at all. The combined language can be effectively hidden from the user by performing the model migration operation sparingly for converting the models written in one of the original DSLs back and forth to the combined DSL when needed. For example, before executing the combined transformation model the instance model could be converted to the combined language and when this transformation finished its execution the instance is converted back to its original language. This allows the transformation to be applicable while the user is oblivious to how it was applicable and to any auxiliary entities (classes, attributes, etc.) that the combined transformation requires.

5.3 Limitations of the Approach

This section presents some of the limitations of our approach that we have identified.

- The DSLs that participate in the combination have to be defined using the *SimpleClassDiagram*, *MoTif*, *TransformationRule* and *ConcreteSyntax* formalisms of AToMPM. Our prototype currently does not support DSLs that have been created using other formalisms;
- The concrete syntax combination is simple and it leaves the detection and resolution of conflicts to the user. Visual concrete syntax combination is not a subject we could find much literature on, with the exception of [27].;
- Our technique does not completely combine the semantics of the DSLs, since the user has to manually combine the combined transformation schedule into an existing one in order to complete the combination. For example, when the RPG DSL instantiates the Pathfinding Template DSL, the semantics of Pathfinding *can* be applied to RPG, but they are not yet completely combined. This has to be done by the user by manually creating a new (or modifying an existing) transformation schedule that applies the semantics of RPG as well as those of Pathfinding. A possible way to include this in our approach could be to make both the RPG and Pathfinding DSLs (following the same example) Template DSLs and then make them instantiate each other In that case, however, the user would need extra knowledge to apply our technique, namely how to create a Template DSL;
- Our approach does not support the automatic propagation of the changes from any of the participant DSL's to the resultant combined DSL, also known as DSL Evolution [28]. This is a feature that, if present, would provide modularity to the process of designing a DSL using our approach;

- During the *match operation*, for the abstract syntax as well as semantics, there are no constraints on the elements that the user designates. A user might for example bind two classes of incompatible types but our solution would simply proceed as normal and then produce a wrong result. This problem could be solved by adapting the idea of *concepts* (see subsubsection 4.2.2.2) to our solution in order to impose extra requirements on the elements to be matched.

Chapter 6

Conclusion

In this thesis we have introduced a solution for the combination of domain-specific languages. We have applied it on an example of a role-playing game DSL and expanded it with new features by grouping them in separate modules that were later combined.

Concerning our main research question, we have shown that it is possible to combine DSLs in a modular way. We have implemented a prototype for the AToMPM tool that allows the combination of DSLs by combining the models that define their syntax and semantics. The ideas of this prototype can be adapted to other meta-modelling tools since they are based on existing techniques for model combination which have been applied in a variety of different tools.

In previous work [15] we applied an ad-hoc and non-modular approach for creating and extending the RPG DSL (our running example) with new features. In this thesis, we applied a modular solution to this problem. We extended the running example via DSL combination by allowing the user to create separate secondary DSLs (e.g., Pathfinding and EventListener) for the extensions and subsequently combining them into the main DSL (e.g., RPG). This facilitates the addition of new features as separate modules, improving the extensibility of our approach. It also allows the independent modification of existing modules, providing improved maintainability.

In this thesis, we have shown that it is possible to combine all the language components of DSLs, i.e., syntax (abstract and concrete) and semantics. We have achieved this by doing two things:

- Splitting a DSL into a set of coordinated models (see chapter 4), where each one represents a language component, i.e., syntax (abstract and concrete) and semantics;

- Applying variants of existing model-based combination techniques to each corresponding pair of these coordinated models: merge (subsection 4.2.1) for the concrete syntax, templates (subsection 4.2.2) for the abstract syntax and interfacing (subsection 4.2.3) for the semantics.

The combination of two DSLs has been explicitly modelled using transformation rules and schedules. Some secondary aspects of the process have been implemented with code. AToMPM tool is capable of supporting language combination techniques thanks to their choice of modelling everything explicitly (including the definition of a language) and supporting model transformations. In this thesis, we have provided a DSL combination technique that is generic, as long as its definition conforms to the following, or similar, languages:

- The abstract syntax must conform to an object oriented modelling language (containing classes, associations, inheritance, fields) such as class diagrams. Our implementation in AToMPM supports the built-in `SimpleClassDiagram` formalism;
- For the concrete syntax, we support a simple dictionary language where strings representing the name of an element are mapped to their visual representation. AToMPM provides the built-in language `ConcreteSyntax` for this purpose and we therefore support the combination of models that conform to it. Unlike for the abstract syntax where using class diagrams is the de facto choice, there is a lack for a standard language to define the concrete syntax of DSLs. However, our approach should be also applicable to other languages with a similar dictionary structure with names of elements and their representation or icons. Since the combination of the concrete syntax is currently a simple merge, that even allows duplicates, it should also be applicable to textual concrete syntax languages, e.g., the language used in [8] and created in metaDepth;
- Concerning the semantics of the languages, it conforms to a transformation schedule language, which contains rule blocks that refer to models defined in a rule transformation language. These two languages have many implementations in different tools which we can adapt to fit our solution

  In order to apply rule-based transformations on the elements of a given language, the metamodel (i.e., abstract syntax, defined using class diagrams) of that language has to be adapted into a *pattern metamodel*. This is done via a process called *RAMification* [16], that performs the following changes to the original metamodel:

  - Relaxation: remove all abstract classes;
  - Augmentation: add label attributes to be used by the transformation engine;

  - Modification: change the data type of the attributes of all model elements to *String*, so they can express conditions or actions.

With a RAMified pattern metamodel available, transformation rules can be created in the rule language that refer to it. This is done by including pattern elements in the LHS and RHS constructs of the rule language.

In AToMPM we use the `MoTif` [26] formalism for the scheduler language, and the built-in `TransformationRule` language and RAMified DSL for the rules.

6.1 Comparison with related work

In this section we compare the solution presented in this thesis with existing solutions for DSL (and model) combination. In [8], Meyers et al. present a similar solution for the complete combination of DSLs created in the textual meta modelling tool *metaDepth*. Our solution differs in that it was implemented for the visual meta-modelling tool AToMPM which is more accessible to non-programmers because of its visual nature. Unlike their solution, we do not use *concepts* to impose constraints on the participant DSLs, since this was not deemed as relevant as the actual combination for the prototype phase, it is thus left for future work. The absence of concepts and thus constraints on the participant DSLs has the drawback of increasing the possibility of unexpected results such as breaking the functionality or semantics of languages by realizing incorrect bindings. Another difference is that we tried to only perform minimal modifications to the source code of AToMPM, while the authors of [8] have created metaDepth and can modify it with more ease.

In previous work [4], a classification of techniques was presented with a comparison table for the current DSL combination techniques. The table, split in two, can be seen in Figure 6.1 and Figure 6.2.

Our approach has been named *DSL Combination* and is located at the bottom of the table. It has the following characteristics:

- *Both*, since the abstract syntax is combined using templates, which is asymmetric, and the concrete syntax using merge, a symmetric technique;
- It is a *homogeneous* technique since it can only be applied to DSLs created using three specific formalisms (one for each language component);
- Its *domain* are class diagrams for the abstract syntax and rule transformations for the semantics;

- Therefore it supports the combination of both the *structural and behavioural* parts of a language (syntax and semantics);
- It is in the prototype stage and therefore classified as *partial implementation*;
- The combination happens after the user has instantiated the interface rules and not at design time, i.e., the moment of composition is *dynamic*;
- The matching method is *semi-automatic* since the user has to manually specify the binding links for the abstract syntax and semantics while this is not necessary for the concrete syntax, where the matching method is automatic;
- *No conflict resolution* strategy is present in our approach since we have decided to focus more on the actual combination of DSLs and not yet its correctness;
- The approach is *highly modular*, it encourages working with small module DSLs separately and combining them in a straight forward manner.
- It also counts with *low intrusion*, which means that the original DSLs have few or no modifications when present in the combined DSL.

6.2 Future Work

This work can be continued in the following different directions that have been left as future work:

- The creation of a library of common Template DSLs. Combined with the presented technique, this would further facilitate the engineering of new DSLs;
- Some aspects of the supported languages have been left out in our solution. For example structural constraints and actions of the *SimpleClassDiagram* formalism, pivots of *TransformationRule* and several more advanced rule blocks of *MoTif*.
- The combination of the concrete syntax of DSLs. Our implementation has employed a simplified version of the Merge technique that restricts the combination and does not take into account conflicts. This can be further explored to support a more and flexible combination.
- The addition of constraints to the combination process in order to ensure the creation of meaningful results. An example of such constraints is presented in [24], by the name of *concepts*.

- As mentioned in section 5.3, the combination of the semantics of DSLs could be extended to allow the combinations of two templates that instantiate each other. This has the drawback of requiring the user to be more involved in the combination process and its exploration is therefore left as future work.

Technique / Criterion	**Symmetry**	**Uniformity**	**Domain**	**Nature of Models**	**DSL Weaving**	**Concrete Syntax**
Pottinger Merge [18]	Symmetric	Homogeneous	Generic	Both	Syntax	None
Package Merge [19]	Symmetric	Homogeneous	UML	Both	Syntax	None
GME Merge [22]	Symmetric	Homogeneous	CD	Structural	Syntax	None
Barbero Extension [11]	Asymmetric	Homogeneous	Generic	Both	Syntax	None
Template Instantiation [5]	Asymmetric	Homogeneous	CD	Both	Syntax	None
MetaDepth Templates [8]	Asymmetric	Homogeneous	CD	Both	Both	Textual
Parametrization [29]	Asymmetric	Homogeneous	CD	Structural	Both	Visual
Embedding [21]	Asymmetric	Homogeneous	Generic	Both	Both	-
Refinement [5]	Asymmetric	Homogeneous	-	-	Both	-
Interfacing [5]	Both	Heterogeneous	-	-	Both	-
Semantic Adaptation [30]	Asymmetric	Heterogeneous	Generic	Both	Semantics	None
GeKo [31]	Asymmetric	Homogeneous	Generic	Both	Both	None
RAM [32]	Asymmetric	Homogeneous	UML	Both	Syntax	None
MATA [33]	Asymmetric	Homogeneous	UML	Both	Syntax	None
GMCF [34]	Both	Homogeneous	Generic	Structural	Syntax	None
AOAM [35]	Asymmetric	Homogeneous	CD	Both	Syntax	None
WEAVR [36]	Asymmetric	Homogeneous	UML	Both	Syntax	None
Semantic Weaving [37]	Asymmetric	Homogeneous	SD	Behavioural	Semantics	None
DSL Combination	Both	Homogeneous	Generic	Both	Both	Visual

FIGURE 6.1: First part of the overview of the classification of composition techniques

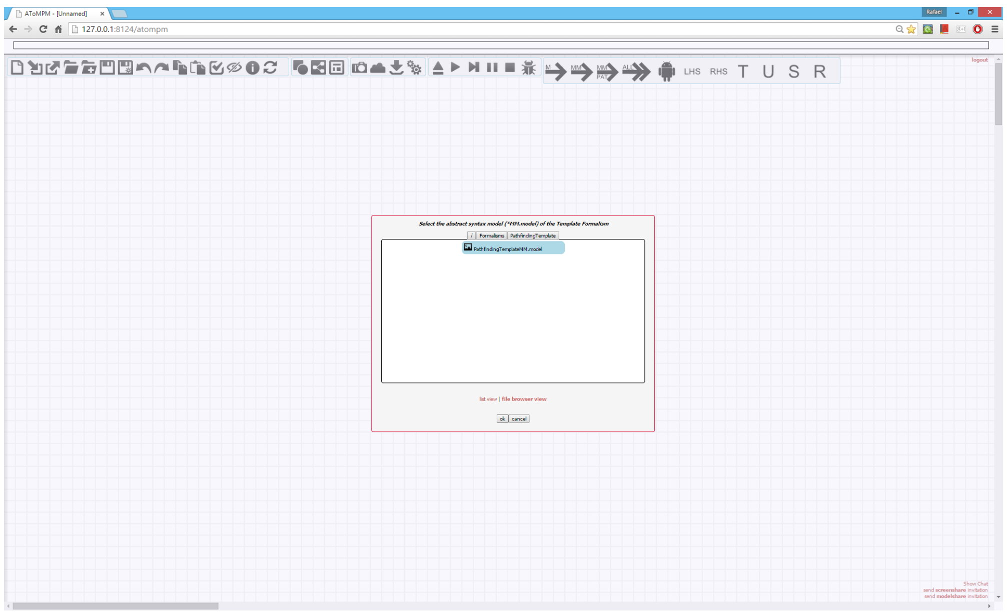
Technique / Criterion	**Implementation**	**Moment of composition**	**Match method**	**Conflict Resolution**	**Modularity**	**Intrusion**
Pottinger Merge [18]	Partial	Static	Manual	Resolution	Low	High
Package Merge [19]	Complete	Static	Automatic	None	Low	Low
GME Merge [22]	Complete	Static	Manual	None	Low	Low
Barbero Extension [11]	Complete	Static	Automatic	None	Low	Low
Template Instantiation [5]	Partial	Static	Manual	None	High	Low
MetaDepth Templates [8]	Complete	Dynamic	Manual	None	High	Low
Parametrization [29]	None	Static	Manual	None	High	Low
Embedding [21]	None	-	-	-	Low	High
Refinement [5]	None	-	-	-	High	Low
Interfacing [5]	None	-	-	-	High	None
Semantic Adaptation [30]	Complete	Dynamic	Manual	None	High	Low
GeKo [31]	Partial	Static	Manual	None	Low	Low
RAM [32]	None	Static	Manual	None	Low	Low
MATA [33]	Partial	Static	Manual	None	Low	Low
GMCF [34]	Partial	Static	Automatic	Resolution	Low	Low
AOAM [35]	Partial	Static	Automatic	Avoidance	Low	Low
WEAVR [36]	Complete	Static	Manual	Avoidance	-	-
Semantic Weaving [37]	Partial	Static	Automatic	Avoidance	Low	High
DSL Combination	Partial	Dynamic	Manual	None	High	Low

FIGURE 6.2: Second part of the overview of the classification of composition techniques

Appendix A

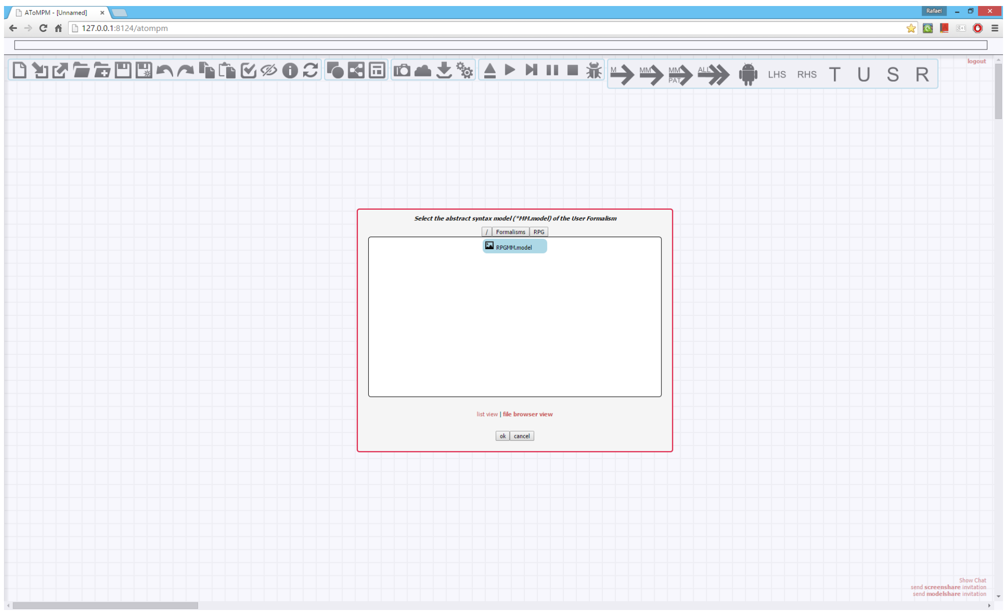
Example Combination in AToMPM

This chapter shows, in screenshots, a full example of the DSL template combination technique in the AToMPM tool.



Select the abstract syntax model (*MM.model) of the Template Formalism

FIGURE A.1: After starting the template combination by pressing the *T* button on the toolbar (top right), the user is presented with this file browser dialog where she is asked to select the desired Template DSL metamodel (see section 5.1). For this example we have chosen the *PathfindintTemplate* DSL



Select the abstract syntax model (*MM.model) of the User Formalism

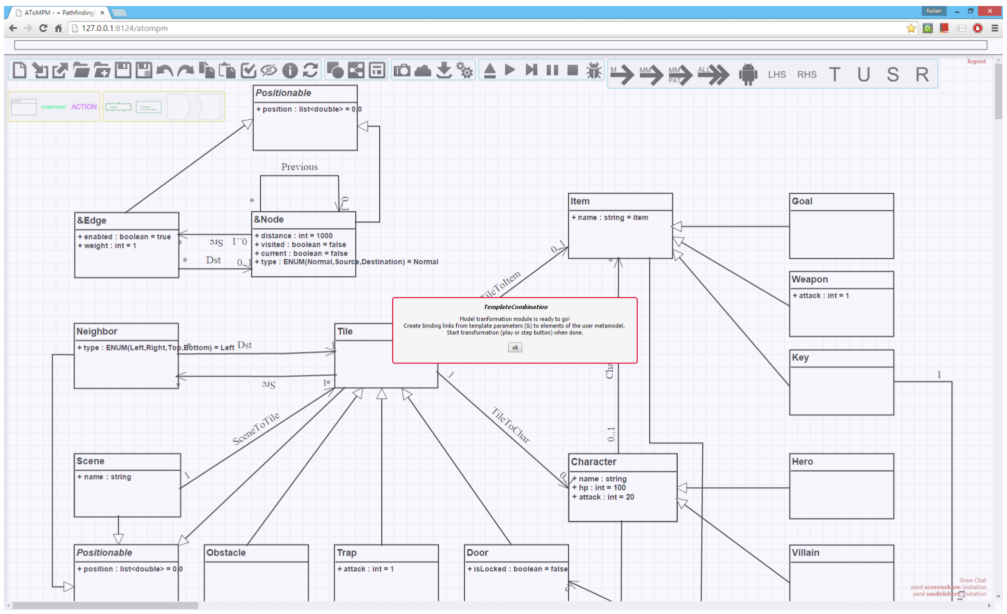
/ Formalisms RPG

RPGMM.model

list view | file browser view

ok cancel

FIGURE A.2: Then, the user is asked to select the User DSL metamodel, in this example the *RPG* DSL.



TemplateCombination

Model tranformation module is ready to go!
Create binding links from template parameters (&) to elements of the user metamodel.
Start transformation (play or step button) when done.

ok

FIGURE A.3: Next, the combination process pre-loads a transformation and asks the user to create the desired binding links (see subsection 5.2.1). When finished, the user can play the already loaded transformation to continue with the combination process.

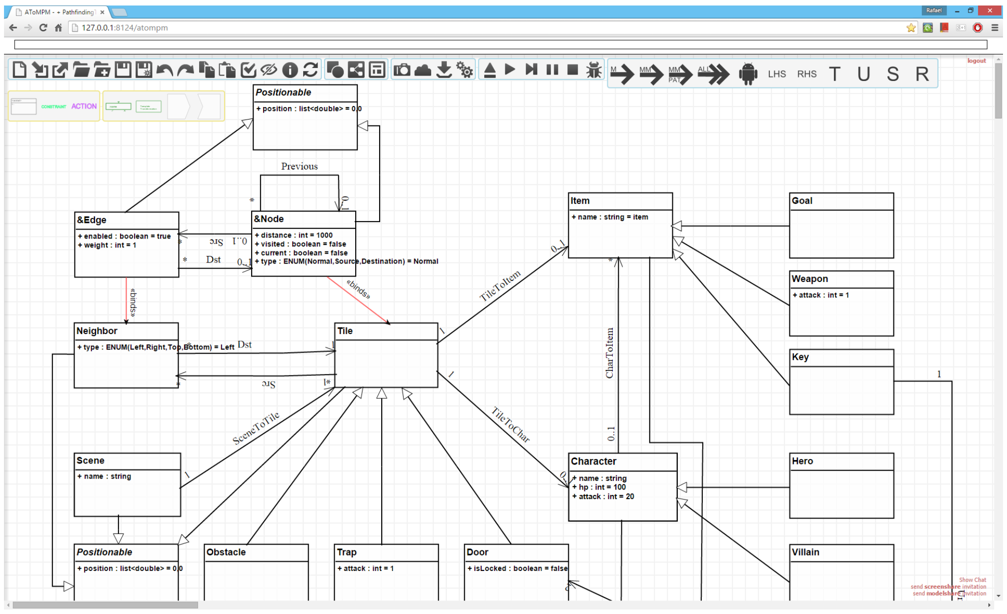
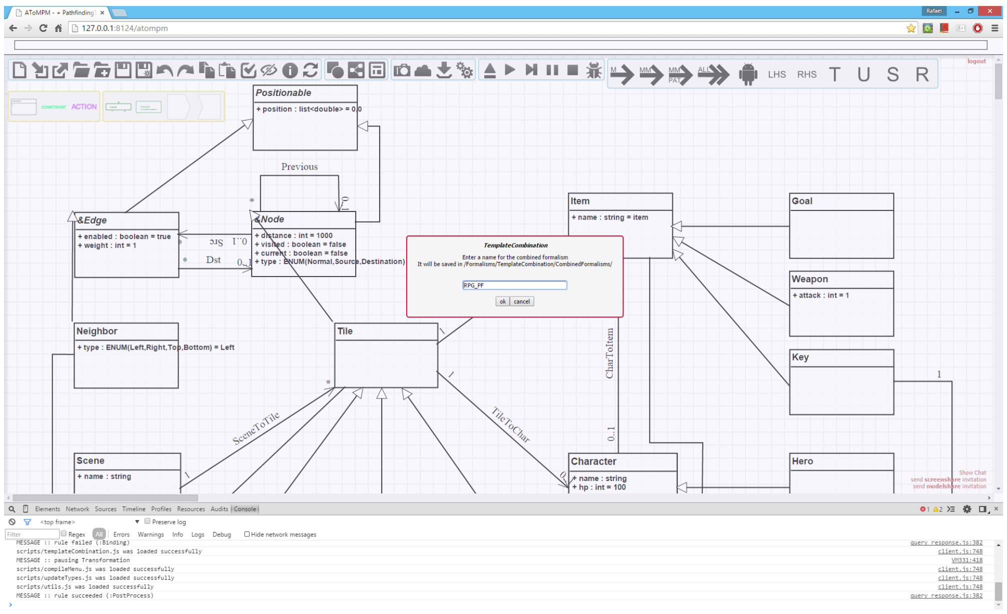


FIGURE A.4: Here, we see the binding links created, coloured red. We have bound the *&Edge* and *&Node* classes to the Neighbour and Tile classes, respectively. This is done with the intention of supporting finding the shortest path from a given Tile to another.


TemplateCombination

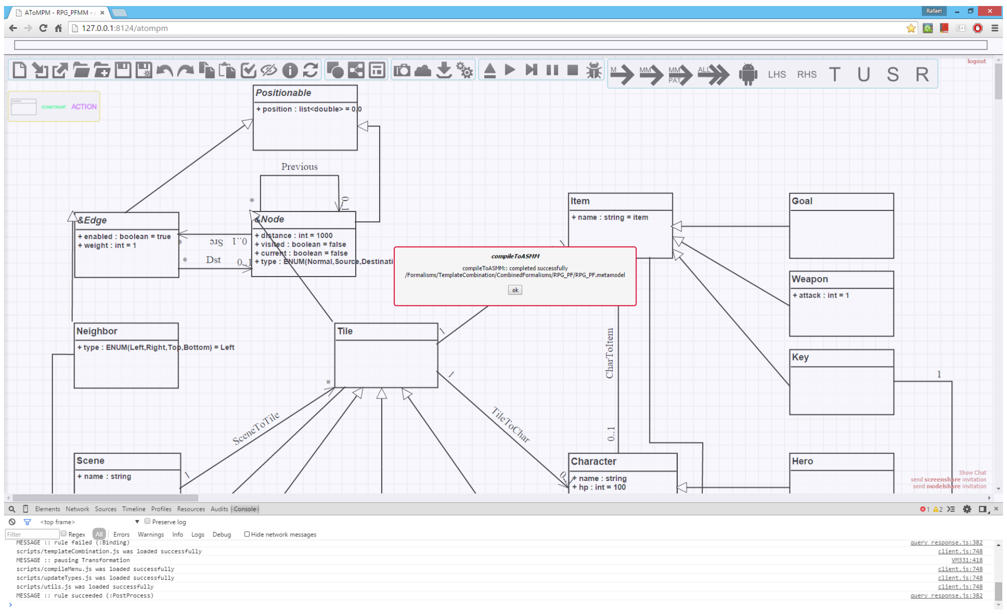
Enter a name for the combined formalism
It will be saved in /Formalisms/TemplateCombination/CombinedFormalisms/

RPG_PF

ok cancel

```
MESSAGE :: rule failed (:Binding)                                   query_response.js:382
scripts/templateCombination.js was loaded successfully              client.js:748
MESSAGE :: pausing Transformation                                   VM331:418
scripts/compileMenu.js was loaded successfully                      client.js:748
scripts/updateTypes.js was loaded successfully                      client.js:748
scripts/utils.js was loaded successfully                            client.js:748
MESSAGE :: rule succeeded (:PostProcess)                            query_response.js:382
```

FIGURE A.5: In this figure we can see the resulting combined metamodel after the *TemplateCombination* transformation has been executed. In this case it replaced the class binding links with inheritance links and it removed the redundant *Src* and *Dst* associations between Tile and Neighbour. The details of this transformation are explained in subsection 5.2.1



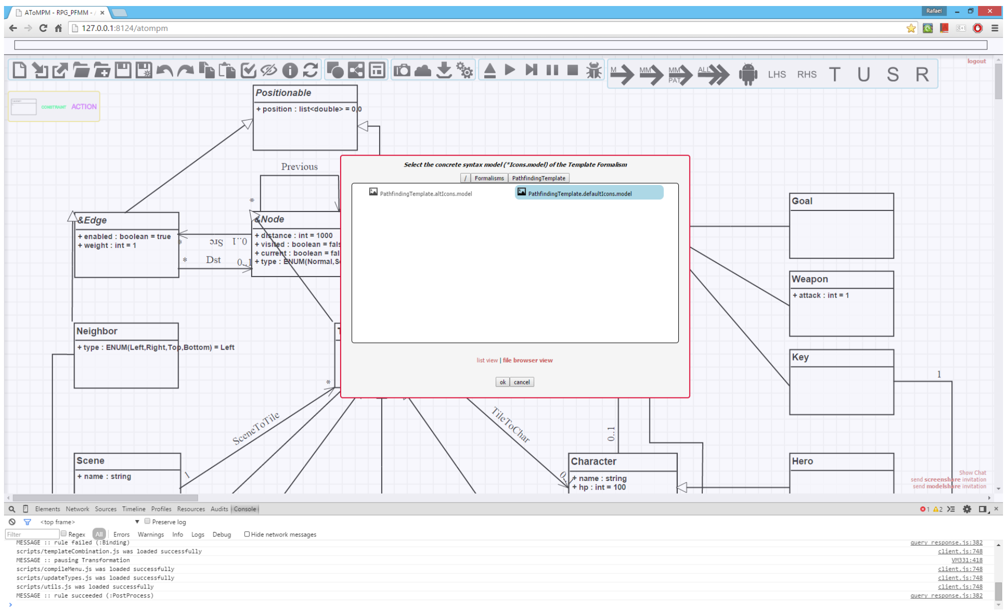
compileToASMM

compileToASMM:: completed successfully
/Formalisms/TemplateCombination/CombinedFormalisms/RPG_PF/RPG_PF.metamodel

ok

```
MESSAGE :: rule failed (:Binding)                              query_response.js:382
scripts/templateCombination.js was loaded successfully         client.js:748
MESSAGE :: pausing Transformation                              VM331:418
scripts/compileMenu.js was loaded successfully                 client.js:748
scripts/updateTypes.js was loaded successfully                 client.js:748
scripts/utils.js was loaded successfully                       client.js:748
MESSAGE :: rule succeeded (:PostProcess)                       query_response.js:382
```

FIGURE A.6: Next, the combined abstract syntax metamodel is compiled into a metamodel file. This file (along with the concrete syntax metamodel file) can be used for loading a metamodel toolbar of the AS which allows the creation of models that conform to the said metamodel.



Select the concrete syntax model (*Icons.model) of the Template Formalism

/ Formalisms PathfindingTemplate

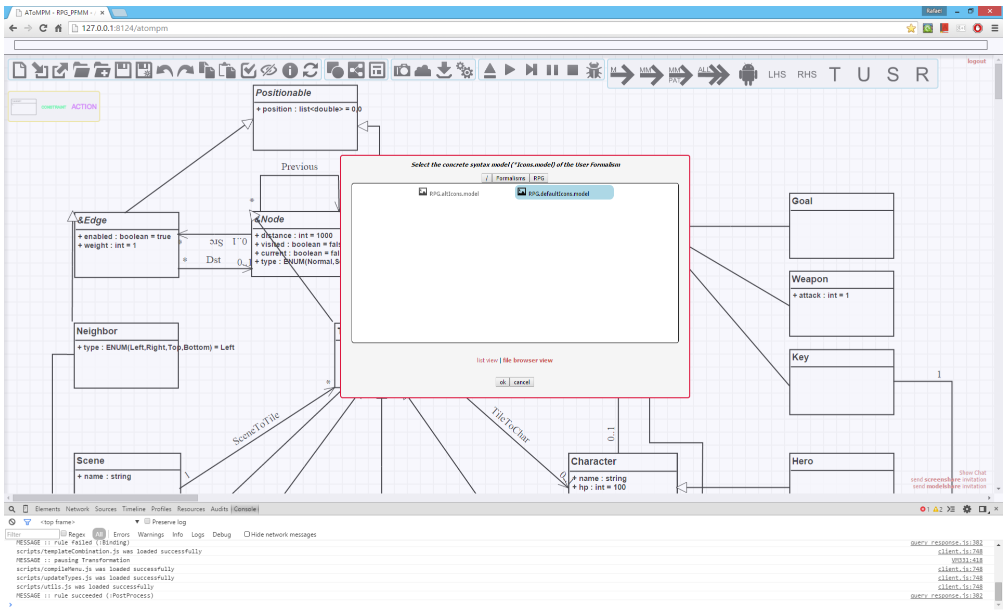
PathfindingTemplate.altIcons.model PathfindingTemplate.defaultIcons.model

list view | file browser view

ok cancel

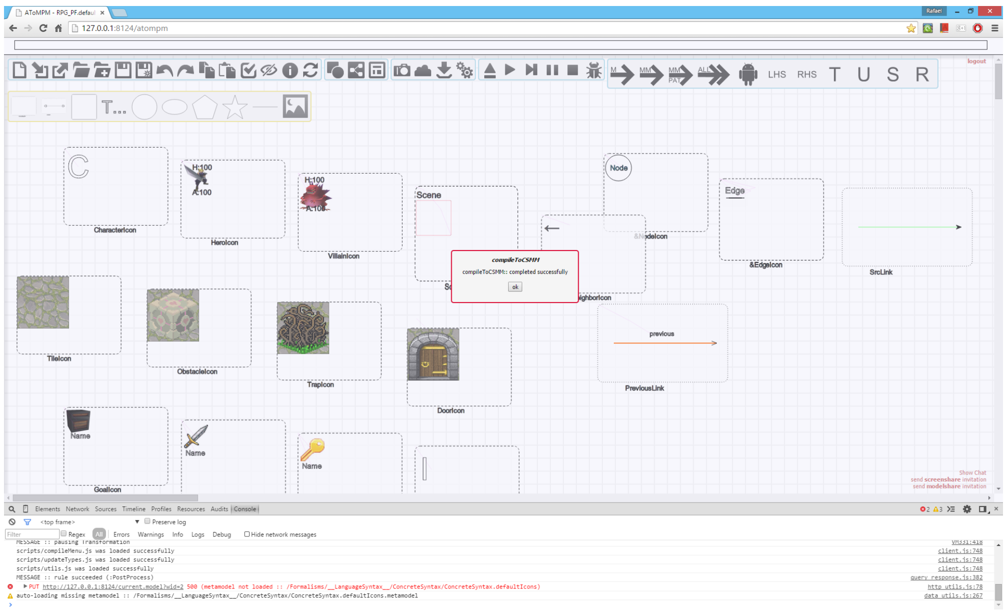
```
MESSAGE :: rule failed (:Binding)                              query_response.js:382
scripts/templateCombination.js was loaded successfully         client.js:748
MESSAGE :: pausing Transformation                              VM331:418
scripts/compileMenu.js was loaded successfully                 client.js:748
scripts/updateTypes.js was loaded successfully                 client.js:748
scripts/utils.js was loaded successfully                       client.js:748
MESSAGE :: rule succeeded (:PostProcess)                       query_response.js:382
```

FIGURE A.7: For combining the concrete syntax, the user is asked to select the desired Template DSL concrete syntax model.



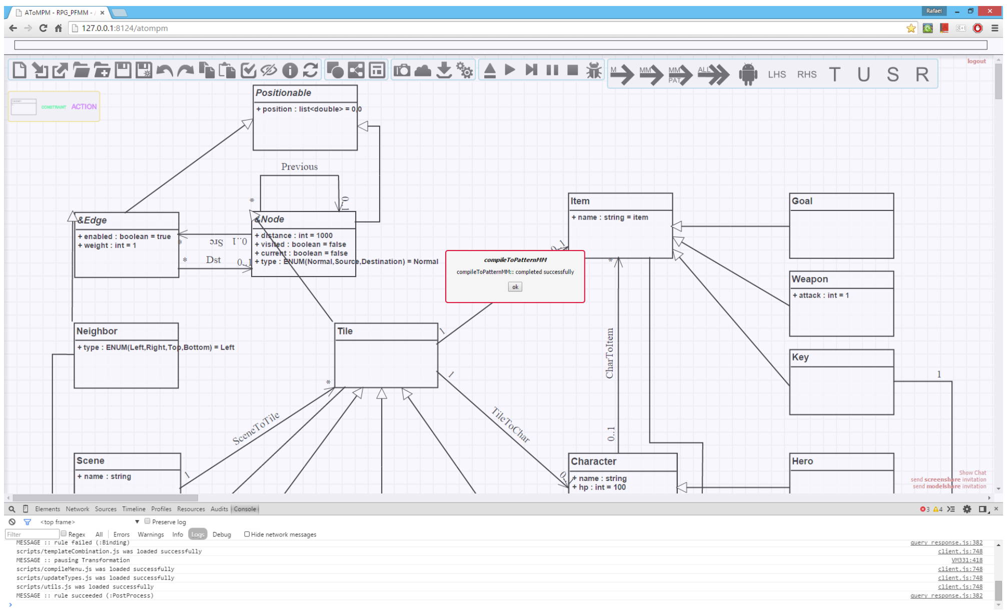
```
MESSAGE :: rule failed (:Binding)                                   query_response.js:382
scripts/templateCombination.js was loaded successfully              client.js:748
MESSAGE :: pausing Transformation                                   VM331:418
scripts/compileMenu.js was loaded successfully                      client.js:748
scripts/updateTypes.js was loaded successfully                      client.js:748
scripts/utils.js was loaded successfully                            client.js:748
MESSAGE :: rule succeeded (:PostProcess)                            query_response.js:382
```

FIGURE A.8: Then, the user is asked to select the desired User DSL concrete syntax model.



```
MESSAGE :: pausing Transformation                                   VM331:418
scripts/compileMenu.js was loaded successfully                      client.js:748
scripts/updateTypes.js was loaded successfully                      client.js:748
scripts/utils.js was loaded successfully                            client.js:748
MESSAGE :: rule succeeded (:PostProcess)                            query_response.js:382
▶PUT http://127.0.0.1:8124/current.model?wid=2 500 (metamodel not loaded :: /Formalisms/__LanguageSyntax__/ConcreteSyntax/ConcreteSyntax.defaultIcons)   http_utils.js:78
auto-loading missing metamodel :: /Formalisms/__LanguageSyntax__/ConcreteSyntax/ConcreteSyntax.defaultIcons.metamodel   data_utils.js:267
```

FIGURE A.9: With the models selected, they are automatically combined as explained in subsection 5.2.2 and subsequently compiled to a concrete syntax metamodel which, together with the abstract syntax metmaodel, allow the creation of instances of the combined DSL.



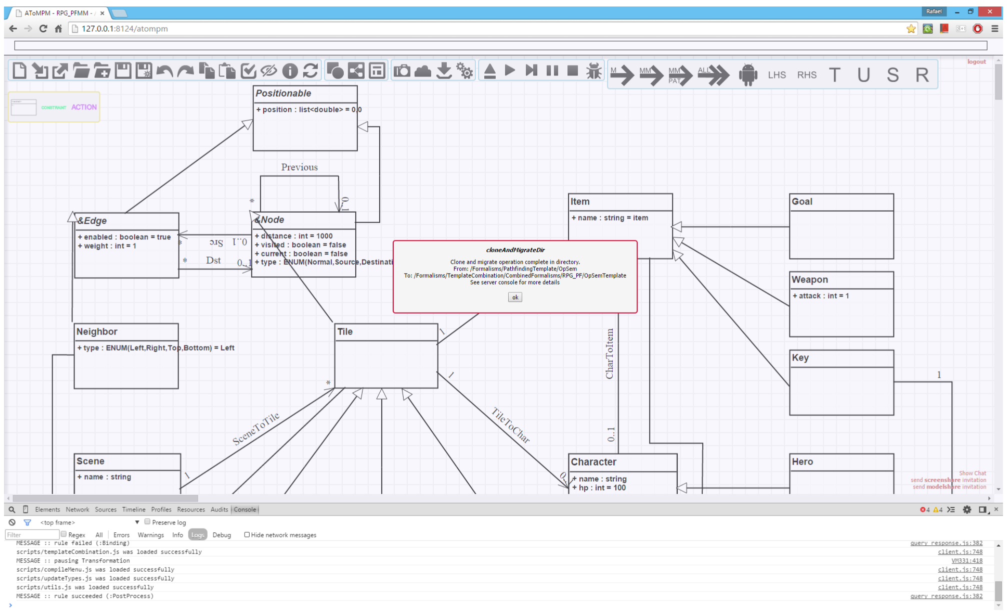
compileToPatternMM

compileToPatternMM:: completed successfully

ok

```
MESSAGE :: rule failed (:Binding)                          query_response.js:382
scripts/templateCombination.js was loaded successfully     client.js:748
MESSAGE :: pausing Transformation                          VM331:418
scripts/compileMenu.js was loaded successfully             client.js:748
scripts/updateTypes.js was loaded successfully             client.js:748
scripts/utils.js was loaded successfully                   client.js:748
MESSAGE :: rule succeeded (:PostProcess)                   query_response.js:382
```

FIGURE A.10: The pattern metamodel is also compiled at this point, since it requires both concrete and abstract syntax model in order to be generated. This pattern metamodel is used in the transformation rules used for defining the operational semantics.



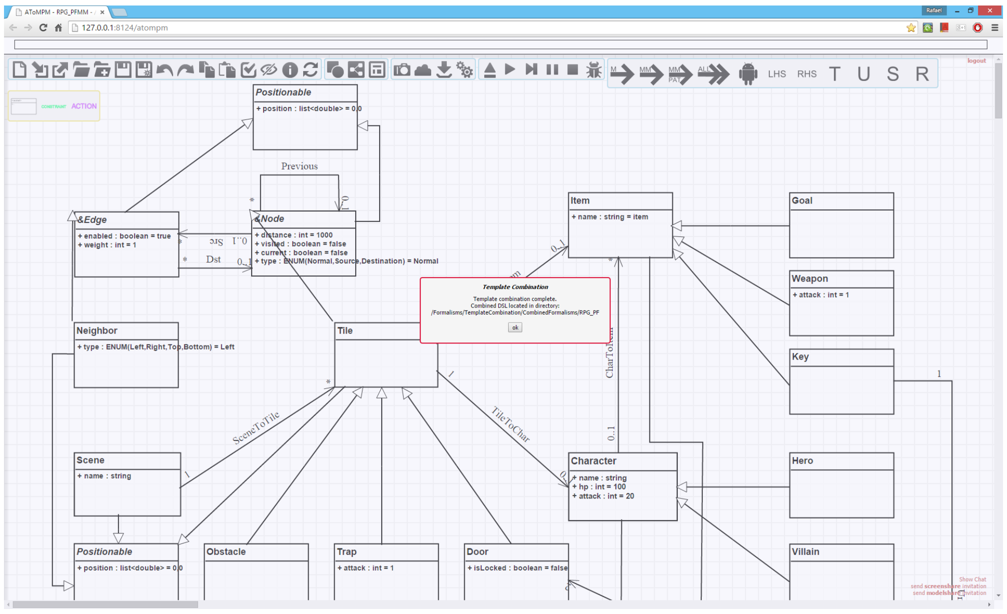
cloneAndMigrateDir

Clone and migrate operation complete in directory.
From: /Formalisms/PathfindingTemplate/OpSem
To: /Formalisms/TemplateCombination/CombinedFormalisms/RPG_PF/OpSemTemplate
See server console for more details

ok

```
MESSAGE :: rule failed (:Binding)                          query_response.js:382
scripts/templateCombination.js was loaded successfully     client.js:748
MESSAGE :: pausing Transformation                          VM331:418
scripts/compileMenu.js was loaded successfully             client.js:748
scripts/updateTypes.js was loaded successfully             client.js:748
scripts/utils.js was loaded successfully                   client.js:748
MESSAGE :: rule succeeded (:PostProcess)                   query_response.js:382
```

FIGURE A.11: Next, the operational semantics models, contained in the *OpSem* folder, are cloned to the folder of the new combined DSL. The models in this folder are also modified, by updating their types, in order to make them compatible with the combined formalism (see subsection 5.2.4).



Template Combination

Template combination complete.
Combined DSL located in directory:
/Formalisms/TemplateCombination/CombinedFormalisms/RPG_PF

ok

FIGURE A.12: After these steps, the guided part of the DSL combination is complete. What still has to be done at this point is to implement the operational semantics interface as explained in subsubsection 5.2.3.1.

Bibliography

[1] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley-IEEE Computer Society Press, 2008.

[2] Douglas C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[3] Raphael Mannadiar and Hans Vangheluwe. Domain-specific engineering of domain-specific languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, page 11. ACM, 2010.

[4] Rafael Ugaz. Weaving of domain-specific languages: A literature review. 2014.

[5] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[6] Bernard P Zeigler, Herbert Praehofer, Tag Gon Kim, et al. *Theory of modeling and simulation*, volume 19. John Wiley New York, 1976.

[7] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 21–28. ACM, 2010.

[8] Bart Meyers, Antonio Cicchetti, Esther Guerra, and Juan De Lara. Composing textual modelling languages in practice. In *Procs. of the Intl. Workshop on Multi-Paradigm Modeling (MPM'12)*, 2012.

[9] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard F Paige. A canonical scheme for model composition. In *Model Driven Architecture–Foundations and Applications*, pages 346–360. Springer, 2006.

[10] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616. ACM, 2006.

[11] Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. A practical approach to model extension. In *Model Driven Architecture-Foundations and Applications*, pages 32–42. Springer, 2007.

[12] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 35–43. ACM, 2005.

[13] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4): 369–385, 2006.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming, in proceedings of the european conference on object-oriented programming (ecoop), finland. *Springer-Verlag LNCS*, 1241:16, 1997.

[15] Rafael Ugaz. Weaving of domain-specific languages: Enabling technology. 2014.

[16] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a model transformation language. In *Domain Engineering*, pages 211–237. Springer, 2013.

[17] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25, 2013.

[18] Rachel A Pottinger and Philip A Bernstein. Merging models based on given correspondences. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 862–873. VLDB Endowment, 2003.

[19] OMG UML. 2.1. 1 superstructure specification (formal/2007-02-03). Technical report, Technical report, Object Management Group, February 2007. available at www. omg. org, downloaded at May 25 th, 2007.

[20] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving uml package merge. *Software & Systems Modeling*, 7(4):443–467, 2008.

[21] Antonio Vallecillo. On the combination of domain specific modeling languages. In *Modelling Foundations and Applications*, pages 305–320. Springer, 2010.

[22] Ákos Lédeczi, Greg Nordstrom, Gabor Karsai, Peter Volgyesi, and Miklos Maroti. On metamodel composition. In *Control Applications, 2001.(CCA'01). Proceedings of the 2001 IEEE International Conference on*, pages 756–760. IEEE, 2001.

[23] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[24] Juan de Lara and Esther Guerra. From types to type requirements: genericity for model-driven engineering. *Software & Systems Modeling*, 12(3):453–474, 2013.

[25] Levi Lucio, Sadaf Mustafiz, J. Denil, B. Meyers, and Vangheluwe H. The formalism transformation graph as a guide to model driven engineering. 2012.

[26] Eugene Syriani and Hans Vangheluwe. A modular timed model transformation language. *Journal on Software and Systems Modeling*, 11:1–28, 2011.

[27] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. In *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pages 889–898. Springer, 2009.

[28] Simon Van Mierlo. Model transformation for modelling language evolution. 2013.

[29] Luis Pedro, Didier Buchs, and Vasco Amaral. Foundations for a domain specific modeling language prototyping environment. 2008.

[30] Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, Hans Vangheluwe, et al. A dsl for explicit semantic adaptation. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling at MODELS 2013*, 2013.

[31] Brice Morin, Jacques Klein, Olivier Barais, and Jean-Marc Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th international workshop on Early Aspects*, pages 11–18. ACM, 2008.

[32] Jacques Klein and Jörg Kienzle. Reusable aspect models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA*, 2007.

[33] Jon Whittle and Praveen Jayaraman. Mata: A tool for aspect-oriented modeling based on graph transformation. In *Models in Software Engineering*, pages 16–27. Springer, 2008.

[34] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. A generic approach for automatic model composition. In *Models in software engineering*, pages 7–15. Springer, 2008.

[35] Y Raghu Reddy, Sudipto Ghosh, Robert B France, Greg Straw, James M Bieman, Nathan McEachen, Eunjee Song, and Geri Georg. Directives for composing aspect-oriented design class models. In *Transactions on Aspect-Oriented Software Development I*, pages 75–105. Springer, 2006.

[36] Thomas Cottenier, Aswin Van Den Berg, and Tzilla Elrad. The motorola weavr: Model weaving in a large industrial context. *Aspect-Oriented Software Development (AOSD), Vancouver, Canada*, 32:44, 2007.

[37] Jacques Klein, Loïc Hélouët, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 27–38. ACM, 2006.